

COURSE

Binary File Formats in Practice

10 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	Bytes and Hex	2
Chapter 2	Integers and Endianness	8
Chapter 3	The struct Module	13
Chapter 4	Magic Numbers and Headers	19
Chapter 5	Reading a PNG	26
Chapter 6	WAV Audio Files	33
Chapter 7	Bit Fields and Flags	41
Chapter 8	Length-Prefixed Records	49
Chapter 9	Robust Parsing	58
Chapter 10	Capstone Container	70

Binary File Formats in Practice

Chapter 1 · Bytes and Hex

Images, audio, archives and documents are all binary files: sequences of bytes with an agreed layout. Once you can read those bytes yourself, a file format stops being a mystery and becomes a specification you can follow. This course builds that skill step by step, starting here with the raw material. All examples use Python 3 with only the standard library, so there is nothing to install.

> **WHAT WAS RUN**
>
> Every snippet in this chapter was executed, and the output shown in the comments is what Python printed. Run them yourself to see the same results.

Bytes Are Small Integers

A byte is a number from 0 to 255. Python's `bytes` type is an immutable sequence of them. A literal looks like a string with a `b` in front, and bytes that are not printable characters are shown as `\x` followed by two hex digits.

```
data = b"Hi!\x00\xff"
print(data)                          # b'Hi!\x00\xff'
print(len(data))                     # 5
print(data[0], data[4])              # 72 255
print(data[0:2])                     # b'Hi'
print(type(data[0]), type(data[0:1]))  # <class 'int'> <class 'bytes'>
print(list(data))                    # [72, 105, 33, 0, 255]
```

Notice the difference between indexing and slicing: `data[0]` is an `int` (72), while `data[0:1]` is a one-byte `bytes` object. Forgetting this is the most common early mistake, for example comparing `data[0] == b"H"`, which is never true.

TYPE	HOLDS	MUTABLE?	EXAMPLE
str	characters (text)	no	`"Hi"`
bytes	bytes, 0 to 255	no	`b"Hi"`
bytearray	bytes, 0 to 255	yes	`bytearray(b"Hi")`

Making Bytes

Besides literals, you can build bytes from a list of integers, from hex text, or by encoding a string. Values outside 0 to 255 are rejected, and a `bytes` object cannot be changed in place.

```
print(data.hex())                    # 48692100ff
print(bytes.fromhex("48 69 21 00 ff"))  # b'Hi!\x00\xff'
print(bytes([72, 105, 33]))          # b'Hi!'
print("é".encode("utf-8"))            # b'\xc3\xa9'

try:
    data[0] = 65
except TypeError as e:
    print("TypeError:", e)          # TypeError: 'bytes' object does not support item as

try:
    bytes([256])
except ValueError as e:
    print("ValueError:", e)         # ValueError: bytes must be in range(0, 256)
```

When you need to change bytes, use a `bytearray`. It behaves like `bytes` but allows assignment, `append` and similar edits. Convert back with `bytes(...)` when you are done.

```
buf = bytearray(data)
buf[0] = 74                      # 74 is the code for "J"
buf.append(10)
print(buf)                       # bytearray(b'Ji!\x00\xff\n')
```

```
print(bytes(buf))                # b'Ji!\x00\xff\n'
```

Reading Hex and Binary

Hexadecimal (base 16) is the standard way to write bytes because one byte is exactly two hex digits, from `00` to `ff`. Python accepts `0x` and `0b` prefixes and has formatting helpers for the other direction.

```
print(0x48, 0xff, 0b1010)          # 72 255 10
print(hex(255), bin(10))            # 0xff 0b1010
print(f"{255:08b}", f"{10:02x}")       # 11111111 0a
```

Files in Binary Mode

Open a file with `"rb"` or `"wb"` and Python gives you `bytes` with no text decoding and no newline translation. `read(n)` returns the next `n` bytes, `tell()` reports your position, and `seek()` moves it. Binary mode matters: opening a binary file in text mode can corrupt data or raise decoding errors.

```
import os, tempfile

path = os.path.join(tempfile.gettempdir(), "demo.bin")
with open(path, "wb") as f:
    n = f.write(bytes(range(0, 40, 3)))
print(n)                           # 14

with open(path, "rb") as f:
    head = f.read(4)
    rest = f.read()
print(head, rest)                  # b'\x00\x03\x06\t' b"\x0c\x0f\x12\x15\x18\x1b\x1e!$'"
print(os.path.getsize(path))       # 14

with open(path, "rb") as f:
    f.seek(-3, 2)                 # 2 means "relative to the end"
```

```
    print(f.read(), f.tell())    # b"!$'" 14
os.remove(path)
```

The odd-looking output is just `repr` choosing its quoting. Byte 9 is a tab, shown as `\t`, and bytes 33, 36 and 39 are the printable characters `!`, `$` and `'`. Because one of them is an apostrophe, Python switched the whole display to double quotes.

Build a Hex Dump

A hex dump shows a file the way a format specification describes it: an offset, the bytes in hex, and a text column where printable characters appear and everything else is a dot. This small function is the tool you will reach for throughout the course.

```
def hexdump(data, width=16):
    lines = []
    for offset in range(0, len(data), width):
        chunk = data[offset:offset + width]
        hexpart = " ".join(f"{b:02x}" for b in chunk)
        text = "".join(chr(b) if 32 <= b < 127 else "." for b in chunk)
        lines.append(f"{offset:08x}  {hexpart:<{width * 3 - 1}}  |{text}|")
    return "\n".join(lines)

sample = b"Binary files: bytes, not text.\n\x00\x01\x02\xff\xfe"
print(hexdump(sample))
```

The output, exactly as printed:

```
00000000  42 69 6e 61 72 79 20 66 69 6c 65 73 3a 20 62 79  |Binary files: by|
00000010  74 65 73 2c 20 6e 6f 74 20 74 65 78 74 2e 0a 00  |tes, not text...|
00000020  01 02 ff fe                                       |....|
```

The first column is the offset in hex, so the second row starts at byte 16 (`0x10`) and the third at byte 32 (`0x20`). In the second row the final `2e` is a real full stop, while the `0a` (a newline) and

`00` after it are not printable, so they show as two more dots, giving `...` after "text". The padding in the last row (`:<{width * 3 - 1}`) keeps the text column lined up when the final chunk is short.

> **A HABIT WORTH FORMING**
>
> Before writing any parsing code, look at the file with a hex dump. Open in `"rb"`, remember that indexing gives an `int` and slicing gives `bytes`, and use a `bytearray` only when you must edit in place.

Challenges

EXERCISE 1

Take the string `"Café"`. Encode it to UTF-8 and print the bytes, print the hex with a space between bytes (look at the optional separator argument of `hex()`), print the length of the string and the length of the bytes, and confirm that decoding gives back the original string. Predict the two lengths first.

View solution

EXERCISE 2

Write every byte value from 0 to 255 once into a temporary file, then read it back in binary mode. Print the length, how many bytes are 128 or greater, and the first four bytes. Then reopen the file, use `seek` to jump to five bytes before the end, and print those five bytes as spaced hex. Delete the file afterwards.

View solution

EXERCISE 3

Write `find_all(data, pattern)` that returns a list of every offset where `pattern` occurs in `data`, including overlapping matches. Use `bytes.find` in a loop. Test it with `b"abcabcab"` and `b"ab"`, with `b"aaaa"` and `b"aa"`, and with a pattern that is not present.

View solution

QUICK REFERENCE

- **bytes** is immutable, **bytearray** is mutable; both hold values 0 to 255
- **Index** gives an `int`; **slice** gives `bytes`
- **Convert:** `.hex()` and `bytes.fromhex()`; `str.encode()` and `bytes.decode()`
- **Hex and binary:** `0x..`, `0b..`, `f"{n:02x}"`, `f"{n:08b}"`
- **Files:** `"rb"` and `"wb"`; `read(n)`, `tell()`, `seek(offset, whence)`
- **Hex dump** your data before you try to parse it

COMING NEXT

Binary File Formats in Practice 2 turns groups of bytes into numbers: how `int.from_bytes` works, what endianness means, and how signed values are stored.

Binary File Formats in Practice

Chapter 2 · Integers and Endianness

Most fields in a binary format are numbers: a length, a width, a count, a timestamp. A number larger than 255 needs more than one byte, which raises two questions every format must answer: which byte comes first, and how are negative numbers stored? This chapter answers both using only Python's built-in integer methods.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed.

From Bytes to an Integer

`int.from_bytes(raw, byteorder)` turns a group of bytes into one integer, and `n.to_bytes(length, byteorder)` does the reverse. You always say how many bytes you want and which order to use. The same two bytes give different answers depending on the order:

```
print(int.from_bytes(b"\x01\x00", "big"))      # 256
print(int.from_bytes(b"\x01\x00", "little"))   # 1
print((258).to_bytes(2, "big"))              # b'\x01\x02'
print((258).to_bytes(2, "little"))           # b'\x02\x01'
print((258).to_bytes(4, "big"))              # b'\x00\x00\x01\x02'
```

The length is a promise about the file layout, not a suggestion. If the number does not fit, Python refuses rather than silently truncating:

```
try:
    (256).to_bytes(1, "big")
except OverflowError as e:
    print("OverflowError:", e)      # OverflowError: int too big to convert
```

ALWAYS PASS THE BYTE ORDER

In recent Python versions the byte order argument defaults to `"big"`, but older versions require it (check which version first made it optional). Writing it explicitly works everywhere and documents your intent, which matters more in file code than saving a few keystrokes.

Big and Little Endian

Endianness is the order in which a multi-byte number's bytes are stored. *Big-endian* puts the most significant byte first, the way we write decimal numbers. *Little-endian* puts the least significant byte first. Neither is wrong; a format simply picks one and every reader must follow it.

```
n = 0x12345678
print(n.to_bytes(4, "big").hex(" "))       # 12 34 56 78
print(n.to_bytes(4, "little").hex(" "))    # 78 56 34 12
print(int.from_bytes(bytes.fromhex("78 56 34 12"), "little") == n)  # True

import sys
print(sys.byteorder)                       # little
```

`sys.byteorder` reports your own machine's native order; it printed `little` here, as it does on common desktop processors (check on yours). Do not use it to decide how to read a file. The file format fixes the order, whatever machine happens to be reading it.

ORDER	FIRST BYTE STORED	0X12345678 ON DISK	SEEN IN
big	most significant	`12 34 56 78`	PNG and many network protocols (check)
little	least significant	`78 56 34 12`	WAV and BMP files (check)

A wrong guess rarely crashes. It just produces a plausible, wrong number. One easy clue is to read both ways and see which result is sensible: a width of 256 pixels is believable, 65536 times that is not.

Signed Values

By default the bytes are treated as unsigned. Most formats store negative numbers in *two's complement*: the top bit of the field acts as a sign, and a pattern with that bit set means the unsigned value minus 2 to the power of the bit count. Pass `signed=True` to both methods to use it.

```
print(int.from_bytes(b"\xff\xfe", "big"))                 # 65534
print(int.from_bytes(b"\xff\xfe", "big", signed=True))    # -2
print((-2).to_bytes(2, "big", signed=True))            # b'\xff\xfe'
print((-2).to_bytes(2, "little", signed=True).hex(" "))  # fe ff

try:
    (-1).to_bytes(1, "big")
except OverflowError as e:
    print("OverflowError:", e)      # OverflowError: can't convert negative int to unsig

for v in (0x7f, 0x80, 0xff):
    print(v, int.from_bytes(bytes([v]), "big", signed=True))
    # 127 127
    # 128 -128
    # 255 -1
```

The same two bytes `ff fe` are 65534 or -2; nothing in the bytes themselves says which. Only the format specification does, so reading it carefully is part of the job. The range of a signed one-byte field is -128 to 127, and the byte `0x80` is the boundary where the sign flips.

Doing It by Hand

Knowing the arithmetic behind `signed=True` helps when you meet odd field sizes or want to check your understanding. Subtract 2 to the power of the bit count when the top bit is set:

```
def to_signed(value, bits):
    if value >= 1 << (bits - 1):
        value -= 1 << bits
    return value

print(to_signed(0xfffe, 16), to_signed(0x7fff, 16), to_signed(0x80, 8))
```

```
# -2 32767 -128
```

THREE QUESTIONS FOR EVERY NUMERIC FIELD

How many bytes is it? Which byte order? Signed or unsigned? A format description that answers all three lets you read the field with one line of Python.

Challenges

EXERCISE 1

Start from `bytes.fromhex("00 00 01 f4 ff ff ff 9c 2a 00")`. Read bytes 0 to 3 as an unsigned big-endian integer, bytes 4 to 7 as a signed big-endian integer, and bytes 8 to 9 as an unsigned little-endian integer, and print the three results. Then print bytes 4 to 7 read as unsigned, and explain the difference from the signed reading.

View solution

EXERCISE 2

Write the values `[1, 256, 65535, 1000]` to a temporary file as two-byte little-endian integers, read the file back and print the raw bytes as spaced hex. Decode the bytes in pairs both as little-endian and as big-endian, print both lists, and check that the little-endian list equals the original. Delete the file afterwards. Which value is unaffected by the wrong order, and why?

View solution

EXERCISE 3

Without using `signed=True`, write `decode_signed(raw)` (big-endian, any length) and `encode_signed(value, size)` using only unsigned conversion and arithmetic. Verify them against `signed=True` for every value from -128 to 127 in one byte, then try `b"\xff\xfe"`, `b"\x80\x00\x00\x00"` and the value -1 in four bytes.

View solution

QUICK REFERENCE

- **Decode:** `int.from_bytes(raw, "big" | "little", signed=False)`
- **Encode:** `n.to_bytes(length, "big" | "little", signed=False)`; raises `OverflowError` if it does not fit
- **Big-endian** stores the most significant byte first; **little-endian** stores the least significant first
- **Two's complement:** top bit set means unsigned value minus 2^{bits}
- **Ask of every field:** size, byte order, signed or unsigned
- `sys.byteorder` is your machine's order, not the file's

COMING NEXT

Binary File Formats in Practice 3 introduces the `struct` module, which reads and writes whole groups of fields at once from a compact format string, including how alignment and padding affect sizes.

Binary File Formats in Practice

Chapter 3 · The struct Module

Reading a header with `int.from_bytes` means slicing the bytes by hand for every field. A real header has a dozen of them. The standard `struct` module describes the whole layout in one short *format string* and converts the entire group in a single call.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed.

Format Strings

A format string is an optional byte-order character followed by one code per field. `struct.pack(fmt, *values)` builds bytes and `struct.unpack(fmt, raw)` returns a tuple, always a tuple, even for one field. Here `H` is an unsigned 16-bit integer and `I` an unsigned 32-bit one:

```
import struct

print(struct.pack("<HI", 1, 2).hex(" "))   # 01 00 02 00 00 00
print(struct.pack(">HI", 1, 2).hex(" "))   # 00 01 00 00 00 02
print(struct.unpack("<HI", b"\x01\x00\x02\x00\x00\x00"))  # (1, 2)
print(struct.calcsize("<HI"))                  # 6
```

The first character picks the byte order, and as in Binary File Formats in Practice 2 it should always be stated for file work: `<` little-endian, `>` big-endian. `struct.calcsize(fmt)` tells you how many bytes a format occupies, which is the number you pass to `f.read()`.

CODE	MEANING	SIZE	PYTHON TYPE
b / B	signed / unsigned byte	1	int
h / H	signed / unsigned short	2	int

CODE	MEANING	SIZE	PYTHON TYPE
i / I	signed / unsigned int	4	int
q / Q	signed / unsigned long long	8	int
f / d	float / double	4 / 8	float
?	boolean	1	bool
4s	fixed-length byte string (4 bytes)	4	bytes
x	one padding byte, no value	1	none

A count before a code repeats it, so `"<3H"` is three unsigned shorts. The exception is `s`, where the count is the length of one byte string rather than a repeat:

```
print(struct.unpack("<3H", b"\x01\x00\x02\x00\x03\x00"))   # (1, 2, 3)
print(struct.pack("<4s", b"ab"))                          # b'ab\x00\x00'
print(struct.pack("<4s", b"abcdef"))                      # b'abcd'
print(struct.pack("<?", True), struct.unpack("<?", b"\x05"))  # b'\x01' (True,)
```

Notice that a short byte string is padded with zero bytes and a long one is silently cut. Neither raises an error, so check lengths yourself when truncation would matter.

A Header in One Call

Suppose a file starts with a 4-byte tag, a 16-bit version, a 16-bit flags value and a 32-bit count, all little-endian. One format describes it, and unpacking gives named variables directly:

```
data = struct.pack("<4sHHI", b"DEMO", 2, 7, 1000)
print(data.hex(" "))                # 44 45 4d 4f 02 00 07 00 e8 03 00 00

magic, ver, flags, count = struct.unpack("<4sHHI", data)
print(magic, ver, flags, count)       # b'DEMO' 2 7 1000
```

Compare that with four slices and four calls to `int.from_bytes`. The format string also doubles as documentation: it is the layout, written once.

Alignment and Padding

The byte-order character has a second job. With `@` (the default when you give none) or `=`, `struct` uses your machine's native rules, and `@` also inserts padding so each field starts at a multiple of its own size, as a C compiler would. The characters `<`, `>`, `!` and `=` use standard sizes; of those, only `=` keeps native byte order, and none of them pad.

```
print(struct.calcsize("@bi"), struct.calcsize("<bi"))   # 8 5
print(struct.pack("@bi", 1, 2).hex(" "))      # 01 00 00 00 02 00 00 00
print(struct.pack("<bi", 1, 2).hex(" "))      # 01 02 00 00 00
print(struct.calcsize("@bxxxi"))             # 8
```

The native results above are from one machine; sizes and padding bytes depend on the platform's rules (check on yours). That is exactly the danger: a file written with `@` is not portable. Padding is also not exclusive to `@` in the file itself: some formats contain padding bytes on purpose, and for those you spell them out with `x`.

> **PREFIX EVERY FILE FORMAT STRING**
>
> Start with `<` or `>` (or `!` for network order, which is big-endian). Then sizes are fixed, there is no hidden padding, and the code behaves the same on every machine.

Offsets, Buffers and Struct Objects

Files are rarely just a header. `unpack_from` reads at an offset without slicing, `pack_into` writes into an existing `bytearray`, and a `struct.Struct` object compiles a format once and gives you the same methods plus `.size`:

```
buf = b"\xaa\xbb" + data + b"\xcc"
print(struct.unpack_from("<4sHHI", buf, 2))   # (b'DEMO', 2, 7, 1000)
```

```
print(struct.unpack_from("<H", buf, 6))        # (2,)

ba = bytearray(12)
struct.pack_into("<HH", ba, 4, 5, 6)
print(ba.hex(" "))                          # 00 00 00 00 05 00 06 00 00 00 00 00

rec = struct.Struct("<HhI")
print(rec.size, rec.format)                # 8 <HhI
blob = rec.pack(1, -1, 70000) + rec.pack(2, -2, 80000)
for t in rec.iter_unpack(blob):
    print(t)
    # (1, -1, 70000)
    # (2, -2, 80000)
```

`iter_unpack` is the natural tool for a table of equal-sized records: it steps through the buffer one record at a time. Because every record is the same size, record number *n* sits at offset *n* times `rec.size`.

When the Bytes Do Not Fit

`struct` is strict. A buffer of the wrong length, a value out of range, or a missing value all raise `struct.error`. Treat it as useful information about the file or your code, not an annoyance:

```
try:
    struct.unpack("<I", b"\x01\x02")
except struct.error as e:
    print("struct.error:", e)   # struct.error: unpack requires a buffer of 4 bytes

try:
    struct.pack("<B", 256)
except struct.error as e:
    print("struct.error:", e)   # struct.error: 'B' format requires 0 <= number <= 255

try:
    struct.pack("<HH", 1)
except struct.error as e:
    print("struct.error:", e)   # struct.error: pack expected 2 items for packing (got 1
```

The exact wording of these messages can differ between Python versions (check).

TASK	CHAPTER 2 TOOLS	STRUCT
One number	`int.from_bytes(raw, "little")`	`struct.unpack("<I", raw)[0]`
Whole header	Slice and convert each field	One format string, one call
Odd sizes (3 bytes)	Works directly	No code for it; use `int.from_bytes`
Table of records	Loop with slicing	`Struct.iter_unpack`

Challenges

EXERCISE 1

For each of the formats `hi`, `bhb`, `bbh`, `qb` and `bq`, print `struct.calcsize` with the `@` prefix and with the `<` prefix. Then print the hex of `struct.pack("@bi", 1, 2)` and `struct.pack("<bi", 1, 2)`, and the size of `"@bxxxi"`. Explain every difference between the two prefixes in terms of padding.

View solution

EXERCISE 2

Using a `struct.Struct("<HhI")` record, write the four records `(1, -5, 1000)`, `(2, 300, 70000)`, `(3, -32768, 4294967295)` and `(4, 0, 0)` to a temporary file, read it back, and print the file length against four times the record size. Check that `iter_unpack` returns the original list, then use `unpack_from` to fetch the third record directly by computing its offset. Delete the file afterwards.

View solution

EXERCISE 3

Define a record `"<8sH"` holding an 8-byte ASCII name and a 16-bit value. Write `pack_record(name, value)` and `unpack_record(raw)` so that the name comes back without its zero padding. Show the packed hex for `"temp"`, round-trip an exactly 8-character name, show what happens to an 11-character name, and catch the error raised when only 5 bytes are supplied.

View solution

QUICK REFERENCE

- **Build / parse:** `struct.pack(fmt, ...)`, `struct.unpack(fmt, raw)` (always a tuple)
- **Size:** `struct.calcsize(fmt)` or `Struct(fmt).size`
- **Prefix:** `<` little, `>` big, `!` network; avoid bare or `@` formats for files
- **Codes:** `B H I Q` unsigned, `b h i q` signed, `f d` floats, `?` bool, `Ns` bytes, `x` pad
- **At an offset:** `unpack_from(fmt, buf, offset)`, `pack_into(fmt, buf, offset, ...)`
- **Tables:** `Struct.iter_unpack(buf)`; record *n* starts at *n* times the size
- **Errors:** `struct.error` for wrong length, out-of-range or missing values

COMING NEXT

Binary File Formats in Practice 4 puts these tools to work on real files: magic numbers that identify a format, how headers are laid out, and how version fields let a reader decide whether it understands what it has been given.

Binary File Formats in Practice

Chapter 4 · Magic Numbers and Headers

A file name is only a label. Rename a text file to `photo.png` and nothing about its bytes changes. Programs that need to know what a file really is look at its first few bytes, where most binary formats put a fixed *signature*, often called a *magic number*. After the signature comes the *header*: the fields that describe everything that follows. This chapter uses the `struct` tools from Binary File Formats in Practice 3 to read both.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed.

Magic Numbers

A magic number is a short, fixed byte sequence at a known position, nearly always offset 0. It is chosen to be unlikely to appear at the start of anything else. Some are readable text, some deliberately are not:

FORMAT	SIGNATURE (HEX)	AS TEXT
PNG	`89 50 4e 47 0d 0a 1a 0a`	`.PNG` then CR LF, Ctrl-Z, LF
JPEG	`ff d8 ff`	not printable
GIF	`47 49 46 38 37 61` or `...39 61`	`GIF87a` / `GIF89a`
PDF	`25 50 44 46 2d`	`%PDF-`
ZIP	`50 4b 03 04`	`PK` then two control bytes
gzip	`1f 8b`	not printable

These values are widely documented but worth confirming against each format's specification before you rely on them for anything important (check). The PNG signature is a good example of careful design: the high-bit first byte stops it being mistaken for text, the letters name the

format, and the CR LF, Ctrl-Z and LF bytes make common text-mode file transfers that mangle line endings show up as a broken signature.

Identifying a File by Its Signature

The technique is a table of signatures and a loop with `bytes.startswith`. Order matters when one signature could begin another, so put the longest and most specific first:

```
SIGNATURES = [
    (b"\x89PNG\r\n\x1a\n", "PNG image"),
    (b"\xff\xd8\xff", "JPEG image"),
    (b"GIF87a", "GIF image"),
    (b"GIF89a", "GIF image"),
    (b"%PDF-", "PDF document"),
    (b"PK\x03\x04", "ZIP archive"),
    (b"\x1f\x8b", "gzip data"),
]

def identify(head):
    for magic, name in SIGNATURES:
        if head.startswith(magic):
            return name
    return "unknown"

print(identify(b"\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR"))   # PNG image
print(identify(b"%PDF-1.7\n%..."))                       # PDF document
print(identify(b"hello world"))                          # unknown
print(identify(b""))                                     # unknown
```

With a real file you only need to read as many bytes as your longest signature. Sixteen is plenty here, and reading a few bytes is far cheaper than reading the whole file. A file shorter than the signature simply fails to match; `startswith` never raises an error for short input.

```
import os, tempfile
path = os.path.join(tempfile.gettempdir(), "demo_sig.bin")
with open(path, "wb") as f:
    f.write(b"PK\x03\x04" + bytes(20))
```

```
with open(path, "rb") as f:
    head = f.read(16)
print(len(head), identify(head))     # 16 ZIP archive
os.remove(path)

print(b"\x89PNG\r\n\x1a\n".hex(" "))   # 89 50 4e 47 0d 0a 1a 0a
print(b"GIF89a"[:3], b"%PDF-1.7"[5:8])  # b'GIF' b'1.7'
```

> **A SIGNATURE IS EVIDENCE, NOT PROOF**
>
> A matching signature only says the file *starts* like that format. A truncated or corrupt file can still match, and a hostile file can be built to. Identify first, then validate the rest of the structure as you parse it (more on this in Binary File Formats in Practice 9).

What a Header Holds

Right after the signature a header usually carries a handful of fixed-size fields. The common ones are:

- **Version:** which revision of the format wrote the file.
- **Flags:** optional features switched on or off (Binary File Formats in Practice 7 takes these apart bit by bit).
- **Counts or sizes:** how many records follow, or how long the data is.
- **Offsets:** where to find the next section, so a reader can jump straight to it.

A header is a fixed-layout record, so a `struct.Struct` describes it exactly. Here is a small invented format: a 4-byte tag, a major and minor version byte, a 16-bit flags value and a 32-bit record count, 12 bytes in all. The parser checks each assumption before trusting the next:

```
import struct

HEADER = struct.Struct("<4sBBHI")
MAGIC = b"DEMO"

def build(major, minor, flags, count):
    return HEADER.pack(MAGIC, major, minor, flags, count)
```

```
def parse_header(raw):
    if len(raw) < HEADER.size:
        raise ValueError(f"header needs {HEADER.size} bytes, got {len(raw)}")
    magic, major, minor, flags, count = HEADER.unpack_from(raw)
    if magic != MAGIC:
        raise ValueError(f"bad magic {magic!r}")
    if major != 1:
        raise ValueError(f"unsupported major version {major}")
    return {"major": major, "minor": minor, "flags": flags, "count": count}

print(HEADER.size)                    # 12
h = build(1, 3, 5, 42)
print(h.hex(" "))                   # 44 45 4d 4f 01 03 05 00 2a 00 00 00
print(parse_header(h))                # {'major': 1, 'minor': 3, 'flags': 5, 'count': 42

for bad in (b"DEMO", b"XXXX" + h[4:], build(2, 0, 0, 0)):
    try:
        parse_header(bad)
    except ValueError as e:
        print("ValueError:", e)
        # ValueError: header needs 12 bytes, got 4
        # ValueError: bad magic b'XXXX'
        # ValueError: unsupported major version 2
```

The order of the checks is deliberate: enough bytes, then the right signature, then a version this code understands. Only after all three does the count mean anything.

Versions

Formats change. A version field lets a reader decide, before it misreads anything, whether it can handle a file. A common convention splits the version in two:

- **Major** changes when the layout changes in a way old readers cannot cope with. A reader refuses a major version it does not know.
- **Minor** changes when something is added in a way old readers can safely ignore. A reader accepts any minor version of a major it knows.

That is why `parse_header` above rejected major 2 but happily accepted minor 3. Refusing clearly is much better than reading garbage and carrying on.

Header Length: Room to Grow

A fixed header size is a trap: add one field and every old reader breaks. Many formats instead store the header's own length, so a reader takes the fields it understands and uses the length to jump to the payload. Fields added later simply sit in the part it skips:

```
V1 = struct.Struct("<4sBBH")

def build2(extra=b""):
    hl = V1.size + 2 + len(extra)
    return V1.pack(b"DEMO", 1, 0, 0) + struct.pack("<H", hl) + extra

def read_header(raw):
    magic, major, minor, flags = V1.unpack_from(raw)
    (hl,) = struct.unpack_from("<H", raw, V1.size)
    return hl, raw[hl:]

a = build2() + b"PAYLOAD"
b = build2(b"\x01\x02\x03\x04") + b"PAYLOAD"
print(read_header(a), read_header(b))   # (10, b'PAYLOAD') (14, b'PAYLOAD')
print(a.hex(" "))   # 44 45 4d 4f 01 00 00 00 0a 00 50 41 59 4c 4f 41 44
print(b.hex(" "))   # 44 45 4d 4f 01 00 00 00 0e 00 01 02 03 04 50 41 59 4c 4f 41 44
```

Both files are read by the same code even though the second has four extra header bytes. In real files, treat a stored length as a claim to be checked: it must be at least the size you need and no larger than the data you actually have.

DESIGN CHOICE	BENEFIT	COST
Magic number	Identify the format without trusting the name	A few bytes of every file
Major / minor version	Readers can refuse or accept safely	Everyone must follow the convention
Header length field	New fields do not break old readers	One more value to validate
Offset to data	Jump straight to a section	Offsets can point outside the file

Challenges

EXERCISE 1

Write `identify_file(path)` using the signature table from this chapter. It reads at most 16 bytes and returns the format name, `"unknown"`, or `"empty file"`. In a temporary folder create a PNG-like file, a PDF-like file, a GIF-like file, a plain text file, a text file named `e.png`, an empty file, a real ZIP made with `zipfile` and a real gzip file made with `gzip`. Print the result for each, and confirm the folder is gone afterwards.

View solution

EXERCISE 2

Using the 12-byte `"<4sBBHI"` header, write `write_demo(path, major, minor, flags, records)` (records are unsigned 16-bit values written after the header) and `read_demo(path)`, which returns `((major, minor), flags, values)`. Reject a truncated header, a wrong magic, a major version other than 1, and a body shorter than the count claims. Show two good files (minor versions 0 and 7) and all four failures, then delete the temporary file.

View solution

EXERCISE 3

Design a header of `"<4sBBH"` followed by a 16-bit `header_length` (10 bytes minimum). Write `write_file(extra, payload)` that sets the length correctly and `read_file(raw)` that returns the flags, the header length and the payload. Show one file with no extra header bytes and one with a 4-byte extra field, read both with the same function, and reject lengths that are too small or larger than the data.

View solution

QUICK REFERENCE

- **Signature check:** read the first bytes, then `head.startswith(magic)`
- **Read only what you need:** `f.read(16)` is enough for the table in this chapter
- **Header as a record:** a module-level `struct.Struct("<...")` with `.size` and `unpack_from`
- **Check order:** enough bytes, then magic, then version, then trust counts and lengths
- **Versions:** refuse an unknown major version; accept any minor of a known major

- **Header length field:** read what you know, skip to the stored length, and validate it first
- **Remember:** a matching signature is evidence, not proof

COMING NEXT

Binary File Formats in Practice 5 reads a real format end to end: the PNG signature, its chunk structure, the length fields, and how a CRC from `zlib` proves each chunk arrived intact.

Binary File Formats in Practice

Chapter 5 · Reading a PNG

The previous chapters built the toolkit: bytes, integers and byte order, `struct`, signatures and headers. A PNG image is a good first real format because it uses all of them and is simple to walk. A PNG is a fixed 8-byte signature followed by a sequence of *chunks*, each with the same layout, and each protected by a checksum. This chapter writes a tiny PNG by hand, reads it back chunk by chunk, and checks every length and CRC along the way. It builds on Binary File Formats in Practice 3 and Binary File Formats in Practice 4.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed.

The Signature

Every PNG starts with the same eight bytes, the signature from Binary File Formats in Practice 4. Nothing else in the file is at a guessable position, so the reader checks this first and refuses anything else:

```
import struct, zlib

SIG = b"\x89PNG\r\n\x1a\n"
print(len(SIG), SIG.hex(" "))    # 8 89 50 4e 47 0d 0a 1a 0a
```

The Chunk Layout

After the signature, everything is a chunk. Each one has four parts, and the integers are **big-endian** (network order), the opposite of the little-endian headers in the last chapter:

PART	SIZE	MEANING
Length	4 bytes	Size of the data part only (not itself, the type or the CRC)
Type	4 bytes	Four ASCII letters, such as `IHDR`, `IDAT`, `IEND`
Data	Length bytes	Chunk contents, which may be empty
CRC	4 bytes	Checksum of the type and the data (not the length)

So a chunk always occupies `length + 12` bytes, and the next chunk starts right after it. The three chunks every simple PNG needs are `IHDR` (the image header, always first), one or more `IDAT` (the compressed pixel data) and `IEND` (always last, with no data). The specification defines more, such as palettes and text; this chapter ignores them (check against the specification for the full list).

CRC With zlib

A CRC (cyclic redundancy check) is a 32-bit fingerprint of some bytes. If even one bit changes, the fingerprint almost certainly changes, so a reader can detect accidental damage. PNG uses the same CRC-32 that `zlib.crc32` computes. It is not a security feature: anyone who edits a chunk can simply recompute the CRC.

Writing a chunk is therefore three `struct.pack` pieces and one checksum. The empty `IEND` chunk is a handy test, because its bytes are the same in every PNG:

```
def chunk(ctype, data=b""):
    body = ctype + data
    return struct.pack(">I", len(data)) + body + struct.pack(">I", zlib.crc32(body))

print(zlib.crc32(b"IEND"), hex(zlib.crc32(b"IEND")))   # 2923585666 0xae426082
print(chunk(b"IEND").hex(" "))                         # 00 00 00 00 49 45 4e 44 ae 42 60 8
```

Read the output left to right: a zero length, the letters `IEND`, and the CRC `ae 42 60 82`. Those last four bytes are the same at the end of every PNG you will ever open.

Building a Tiny Image

The `IHDR` data is 13 bytes: width and height (4 bytes each), then five single bytes: bit depth, colour type, compression method, filter method and interlace method. Colour type 0 means greyscale and bit depth 8 gives one byte per pixel. The pixel data itself is each row preceded by a one-byte *filter type* (0 means "none"), then the whole stream compressed with zlib and stored in `IDAT`:

```
def make_png(rows):
    h, w = len(rows), len(rows[0])
    ihdr = struct.pack(">IIBBBBB", w, h, 8, 0, 0, 0, 0)
    raw = b"".join(b"\x00" + bytes(r) for r in rows)
    return SIG + chunk(b"IHDR", ihdr) + chunk(b"IDAT", zlib.compress(raw)) + chunk(b"IEN

png = make_png([[0, 64, 128, 255], [255, 128, 64, 0]])
print(png[:8] == SIG, png[8:33].hex(" "))
# True 00 00 00 0d 49 48 44 52 00 00 00 04 00 00 00 02 08 00 00 00 00 5a c3 22 bf
```

Take that hex apart with what you know. `00 00 00 0d` is length 13. `49 48 44 52` is the ASCII for `IHDR`. Then width 4, height 2, depth 8, and four zero bytes. The final four bytes, `5a c3 22 bf`, are the CRC of the type and the 13 data bytes.

Walking the Chunks

A reader repeats one step: read the length and type, check the chunk fits inside the file, slice out the data, read the stored CRC and compare. A generator keeps the loop tidy. Notice that every length is checked against the real data size before it is used, the habit that Binary File Formats in Practice 9 develops fully:

```
def read_chunks(raw):
    if raw[:8] != SIG:
        raise ValueError("not a PNG: bad signature")
    pos = 8
    while pos < len(raw):
```

```
        if pos + 8 > len(raw):
            raise ValueError("truncated chunk header")
        length, ctype = struct.unpack_from(">I4s", raw, pos)
        end = pos + 12 + length
        if end > len(raw):
            raise ValueError(f"chunk {ctype!r} runs past end of file")
        data = raw[pos + 8:pos + 8 + length]
        (stored,) = struct.unpack_from(">I", raw, pos + 8 + length)
        if stored != zlib.crc32(ctype + data):
            raise ValueError(f"CRC mismatch in {ctype!r}")
        yield ctype, data
        pos = end

for ctype, data in read_chunks(png):
    print(ctype, len(data))
# b'IHDR' 13
# b'IDAT' 18
# b'IEND' 0
```

The `IDAT` size of 18 is what this machine's zlib produced for this tiny image; another zlib version or setting may give a different number, but the chunk structure is identical (check).

Reading the Header and the Pixels

Collecting the chunks into a dictionary is fine for this simple file (a real PNG may have several `IDAT` chunks whose data must be joined before decompressing, as Challenge 3 does). Unpacking `IHDR` gives the image properties, and `zlib.decompress` reverses the compression of the pixel data:

```
chunks = dict(read_chunks(png))
w, h, depth, color, comp, filt, interlace = struct.unpack(">IIBBBBB", chunks[b"IHDR"])
print(w, h, depth, color, comp, filt, interlace)   # 4 2 8 0 0 0 0

raw = zlib.decompress(chunks[b"IDAT"])
print(raw.hex(" "))                 # 00 00 40 80 ff 00 ff 80 40 00

stride = 1 + w                      # one filter byte plus one byte per pixel
print([list(raw[y * stride + 1:(y + 1) * stride]) for y in range(h)])
```

```
# [[0, 64, 128, 255], [255, 128, 64, 0]]
```

The two rows come back exactly as they went in. This only works because every row used filter type 0. Real encoders usually choose other filters (the specification lists five) that store each pixel relative to its neighbours, because that compresses better. Decoding those is more work, so a robust reader checks the filter byte and refuses what it does not support, as Challenge 3 does.

Catching Damage

Flip a single bit inside the `IHDR` data and the stored CRC no longer matches, so the reader stops before it ever trusts the width:

```
bad = bytearray(png)
bad[19] ^= 0x01          # last byte of the width field
try:
    list(read_chunks(bytes(bad)))
except ValueError as e:
    print("ValueError:", e)   # ValueError: CRC mismatch in b'IHDR'
```

CHECK THE CRC FIRST, THEN INTERPRET

Verify a chunk's CRC before unpacking its fields. A corrupt length or width can make later code allocate huge buffers or loop for ages; failing at the CRC keeps bad numbers out of the rest of the program. Remember the limits: a CRC catches accidental damage, not deliberate tampering.

IDEA	LAST CHAPTER'S DEMO HEADER	PNG
Byte order	Little-endian (`<`)	Big-endian (`>`)
Signature	4 bytes of text	8 bytes designed to survive text-mode mangling
Structure	One fixed header	Repeating length-type-data-CRC chunks
Damage detection	None	CRC-32 per chunk

Challenges

EXERCISE 1

Write `list_chunks(raw)` that walks a PNG and returns a list of `(type, length)` pairs without checking CRCs. It must reject a wrong signature, a chunk whose length runs past the end of the data, and a file with too few bytes left to hold a chunk header. Build a 3 by 1 greyscale PNG by hand and show the list plus all three failures.

View solution

EXERCISE 2

Write `verify(raw)` that returns a list of `(type, ok)` pairs, where `ok` says whether each chunk's stored CRC matches. Show a clean 2 by 2 image, then the same image with one bit of the `IHDR` width flipped, then with the stored CRC byte itself damaged, and finally with an `IDAT` length byte damaged (which should raise an error, not report a CRC problem). Explain in a comment why the last case differs.

View solution

EXERCISE 3

Write `write_gray_png(path, rows)` and `read_gray_png(path)` for 8-bit greyscale images. The reader checks every CRC, joins all `IDAT` data, rejects any image that is not 8-bit greyscale and non-interlaced, rejects any row with a filter other than 0, and returns the rows. In a temporary folder, write a 4 by 3 image, read it back and compare, then damage one byte inside `IHDR` and show the error. Remove the folder and confirm it is gone.

View solution

QUICK REFERENCE

- **Signature:** `b"\x89PNG\r\n\x1a\n"`, 8 bytes, checked first
- **Chunk:** length (4, big-endian) + type (4) + data + CRC (4); total `length + 12`
- **CRC covers:** type and data only, computed with `zlib.crc32`
- **Essential chunks:** `IHDR` first, `IDAT` (one or more), `IEND` last

- **IHDR:** `struct.unpack(">IIBBBBB", data)` gives width, height, depth, colour type, compression, filter, interlace
- **Pixels:** join `IDAT` data, `zlib.decompress`, then each row is a filter byte plus its pixels
- **Always:** check each length against the data you really have, and verify the CRC before using a chunk

COMING NEXT

Binary File Formats in Practice 6 moves from images to sound: the WAV file, a RIFF container of chunks with a format description, and writing your own tone with Python's `wave` module.

Binary File Formats in Practice

Chapter 6 · WAV Audio Files

The PNG in the last chapter was a signature followed by chunks. A WAV file is built the same way, with two differences worth noticing: it wraps everything in an outer *RIFF* container, and its integers are **little-endian** again. Sound is also a gentler subject for practice than images, because "the pixel data" is just a list of numbers over time. This chapter pulls apart a tiny WAV, reads its format fields, generates a tone, and then lets Python's `wave` module do the same job. It builds on Binary File Formats in Practice 2, Binary File Formats in Practice 3 and Binary File Formats in Practice 5.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed. Everything is done in memory or in a temporary folder, so nothing is left behind.

Sound as Numbers

A microphone signal is continuous, so digital audio *samples* it: it measures the level many times per second and stores each measurement as an integer. Three numbers describe the result:

- **Sample rate**: measurements per second (8000 for crude telephone quality, 44100 for CD audio).
- **Bits per sample**: how precisely each level is stored. 16 bits per sample means a signed integer from -32768 to 32767.
- **Channels**: 1 for mono, 2 for stereo. One *frame* holds one sample for every channel.

Storing the samples as they are, with no compression, is called **PCM** (pulse-code modulation). That is the plain kind of WAV this chapter handles; WAV can also hold compressed audio, which we refuse politely rather than decode.

The RIFF Layout

RIFF is a general container: a 4-byte id, a 4-byte size, then contents. A WAV is a RIFF whose form type is `WAVE`, followed by chunks in the same id-size-data shape as PNG, but with a little-

endian size and **no CRC**:

OFFSET	SIZE	FIELD	MEANING
0	4	`RIFF`	Container id
4	4	Size	Bytes after this field, so file size minus 8
8	4	`WAVE`	Form type
12	8 + n	Chunks	Each is id (4), size (4, little-endian), data

There is one rule PNG did not have: if a chunk's size is **odd**, a single pad byte follows its data so the next chunk starts on an even offset. The pad byte is not counted in the size. A reader steps to the next chunk with `pos += 8 + size + (size & 1)`.

Writing a Tiny WAV

Let `wave` write a file of four samples into a `BytesIO` (an in-memory file, so nothing touches the disk), then look at the raw bytes:

```
import struct, wave, io, math

buf = io.BytesIO()
with wave.open(buf, "wb") as w:
    w.setnchannels(1)
    w.setsampwidth(2)           # bytes per sample: 2 = 16 bits
    w.setframerate(8000)
    w.writeframes(struct.pack("<4h", 0, 1000, -1000, 32767))
raw = buf.getvalue()
print(len(raw))                 # 52
print(raw.hex(" "))
# 52 49 46 46 2c 00 00 00 57 41 56 45 66 6d 74 20 10 00 00 00 01 00 01 00 40 1f 00 00 80
```

Read it in pieces. `52 49 46 46` is `RIFF`; `2c 00 00 00` is 44, which is 52 minus 8; `57 41 56 45` is `WAVE`. Then `66 6d 74 20` is `fmt` (note the trailing space, because ids are exactly four bytes) with size `10 00 00 00`, 16. After its 16 bytes comes `64 61 74 61`, `data`, with size 8: four

samples of two bytes each. The last eight bytes are the samples themselves, `00 00`, `e8 03` (1000), `18 fc` (-1000) and `ff 7f` (32767), each little-endian.

The fmt Chunk

The 16 bytes of the `fmt` chunk unpack with one format string, `"<HHIIHH"`:

FIELD	CODE	MEANING	VALUE ABOVE
Format tag	`H`	1 means plain PCM	1
Channels	`H`	1 mono, 2 stereo	1
Sample rate	`I`	Frames per second	8000
Byte rate	`I`	Sample rate times block align	16000
Block align	`H`	Bytes per frame: channels times bytes per sample	2
Bits per sample	`H`	Usually 8, 16 or 24	16

The byte rate and block align can be worked out from the other fields. They are stored anyway, and a careful reader checks that they agree, because a file that contradicts itself is damaged or hostile.

Walking the Chunks by Hand

This is the PNG walker again with three changes: little-endian sizes, the form type check, and the pad byte. The length is checked against the data really present before it is used:

```
  def walk(raw):
      if raw[:4] != b"RIFF" or raw[8:12] != b"WAVE":
          raise ValueError("not a WAV file")
      (riff_size,) = struct.unpack_from("<I", raw, 4)
      if riff_size + 8 != len(raw):
          raise ValueError("RIFF size does not match file size")
      pos = 12
      while pos < len(raw):
          if pos + 8 > len(raw):
```

```
            raise ValueError("truncated chunk header")
        cid, size = struct.unpack_from("<4sI", raw, pos)
        if pos + 8 + size > len(raw):
            raise ValueError(f"chunk {cid!r} runs past end of file")
        yield cid, raw[pos + 8:pos + 8 + size]
        pos += 8 + size + (size & 1)

for cid, data in walk(raw):
    print(cid, len(data))
# b'fmt ' 16
# b'data' 8

chunks = dict(walk(raw))
fmt, ch, rate, brate, align, bits = struct.unpack("<HHIIHH", chunks[b"fmt "])
print(fmt, ch, rate, brate, align, bits)             # 1 1 8000 16000 2 16
print(struct.unpack("<4h", chunks[b"data"]))        # (0, 1000, -1000, 32767)
```

Files written by other programs often contain extra chunks, such as `LIST` (metadata like a title or the encoder name) (check). Because the walker skips by size, it steps over chunks it does not understand, which is how RIFF readers stay compatible. Be careful with the size check on the outer container, though: some real-world writers leave a wrong size in a streamed file (check), so a tolerant reader might only warn. This course's reader is strict.

Generating a Tone

A pure tone is a sine wave. For a frequency `f` in hertz, sample number `i` at rate `r` is at time `i / r` seconds, and the level there is `sin(2 * pi * f * i / r)`. Scale that from the range -1 to 1 up to 16-bit integers, keeping the amplitude below the maximum so the peaks do not clip:

```
def tone_bytes(freq, secs, rate=8000, amp=0.5):
    n = int(rate * secs)
    return b"".join(
        struct.pack("<h", round(amp * 32767 * math.sin(2 * math.pi * freq * i / rate)))
        for i in range(n)
    )

t = tone_bytes(440, 0.01)
print(len(t), struct.unpack("<8h", t[:16]))
```

```
# 160 (0, 5550, 10443, 14102, 16093, 16182, 14357, 10835)
```

A hundredth of a second at 8000 samples per second is 80 samples, so 160 bytes. The values rise to a peak and begin to fall: the first part of a wave. The exact digits depend on floating-point rounding, so a different platform could differ in the last place (check).

Writing the Container Yourself

Writing is easier than reading, because you control every field. This function builds the whole file from PCM bytes, and the test compares it with what the `wave` module produced earlier:

```
def make_wav(pcm, rate=8000, channels=1):
    fmt = struct.pack("<HHIIHH", 1, channels, rate, rate * channels * 2, channels * 2, 1(
    body = b"WAVE" + b"fmt " + struct.pack("<I", 16) + fmt
    body += b"data" + struct.pack("<I", len(pcm)) + pcm
    return b"RIFF" + struct.pack("<I", len(body)) + body

mine = make_wav(struct.pack("<4h", 0, 1000, -1000, 32767))
print(mine == raw)        # True
```

The two files are byte-for-byte identical. The only value computed late is the RIFF size, which needs the length of everything after it, so the body is built first and measured.

Reading Back With wave

For ordinary files the standard library does the parsing. Write half a second of 440 Hz (the musical note A above middle C) and ask the module about it:

```
buf = io.BytesIO()
with wave.open(buf, "wb") as w:
    w.setnchannels(1); w.setsampwidth(2); w.setframerate(8000)
    w.writeframes(tone_bytes(440, 0.5))
```

```
    buf.seek(0)
    with wave.open(buf, "rb") as r:
        print(r.getnchannels(), r.getsampwidth(), r.getframerate(), r.getnframes(), r.getnfr
        # 1 2 8000 4000 0.5
        print(len(r.readframes(10)))     # 20
```

`readframes(10)` returns 10 *frames*, which is 20 bytes here because each mono 16-bit frame is 2 bytes. Frames, not bytes, is the unit the module counts in. The module reads plain PCM only and raises an error for other format tags, which is why a hand-written reader is still worth knowing: you can see exactly what it accepts.

WRITING TO DISK AND PLAYING IT

Pass a file path to `wave.open` instead of a `BytesIO` to save a real `.wav` file, which any audio player can open. The exercises use a temporary folder and remove it afterwards, so you can listen by keeping the file if you want to.

IDEA	PNG (BINARY FILE FORMATS IN PRACTICE 5)	WAV
Outer container	8-byte signature only	`RIFF`, size, `WAVE` (12 bytes)
Byte order	Big-endian	Little-endian
Chunk shape	Length, type, data, CRC	Id, size, data, pad byte if the size is odd
Damage detection	CRC-32 per chunk	None; check sizes and field agreement instead
Payload	zlib-compressed rows	Raw samples

Challenges

EXERCISE 1

Write `list_chunks(raw)` that walks a WAV file and returns a list of `(id, size)` pairs. It must reject a wrong signature, a chunk whose size runs past the end of the data, and a file with too few bytes left to hold a chunk header, and it must step over the pad byte after an odd-sized chunk. Build a WAV by hand

that includes an extra odd-sized chunk before the `data` chunk, show the list, and show all three failures.

View solution

EXERCISE 2

Write `describe(raw)` that returns a dictionary with channels, sample rate, bits per sample, number of frames and duration in seconds. It must refuse a format tag other than 1, a file with no `fmt` or `data` chunk, and a header whose byte rate or block align does not agree with the other fields. Show a one-second mono 8000 Hz file, a one-second stereo 44100 Hz file, and the two header failures.

View solution

EXERCISE 3

Write `write_tone(path, freq, secs, rate, amp)` using the `wave` module and `read_by_hand(path)` that parses the same file with `struct` and returns the rate and the list of samples. In a temporary folder, write a quarter second of 440 Hz, read the samples with `wave` and by hand, and show that the results are equal, that the peak is close to half of full scale, and that the file size is 44 bytes plus 2 bytes per frame. Remove the folder and confirm it is gone.

View solution

QUICK REFERENCE

- **Header:** `RIFF`, little-endian size (file size minus 8), `WAVE`; chunks start at offset 12
- **Chunk:** id (4) + size (4, little-endian) + data, plus one pad byte if the size is odd
- **fmt chunk:** `struct.unpack("<HHIIHH", data)` gives tag, channels, rate, byte rate, block align, bits
- **PCM:** tag 1; frame size = channels times bytes per sample = block align
- **Samples:** 16-bit PCM is signed little-endian, `"<h"` each
- **Duration:** frames divided by sample rate
- **wave module:** `setnchannels`, `setsampwidth`, `setframerate`, `writeframes`; counts in frames, not bytes
- **Always:** check each size against the data you really have, and check that redundant fields agree

COMING NEXT

Binary File Formats in Practice 7 looks inside a single byte: bit fields and flags, using masks and shifts to pack several yes/no values into one number and take them apart again.

Binary File Formats in Practice

Chapter 7 · Bit Fields and Flags

Every field so far has been a whole number of bytes. Real formats are thriftier: a single byte often carries several yes/no answers, or a small number squeezed in beside them, because a header that is read millions of times is worth keeping short. This chapter opens up the byte. It covers the bitwise operators, the standard recipes for testing, setting, clearing and extracting bits, and then reads two real packed fields. It builds on Binary File Formats in Practice 2 and Binary File Formats in Practice 4.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed. Everything is done in memory, so nothing is left behind.

Bits Are Numbered From the Right

A byte has eight bits. **Bit 0** is the least significant, the rightmost one, worth 1; bit 7 is the leftmost, worth 128. Format documents nearly always number bits this way, so "bit 3" means the value 8 (`1 << 3`). Do not confuse this with *byte* order from Binary File Formats in Practice 2: endianness is about the order of whole bytes in a larger number, while bit numbering inside one byte is fixed by place value.

The Operators

OPERATOR	NAME	EFFECT ON EACH BIT
`a & b`	AND	1 only where both are 1: keeps bits
`a | b`	OR	1 where either is 1: turns bits on
`a ^ b`	XOR	1 where they differ: flips bits
`~a`	NOT	Flips every bit
`a << n`	Shift left	Moves bits up n places (multiplies by 2^n)

OPERATOR	NAME	EFFECT ON EACH BIT
`a >> n`	Shift right	Moves bits down n places, dropping the low ones

```
b = 0b10110010
print(bin(b), b)                      # 0b10110010 178
print(bin(b & 0b00001111))            # 0b10          (low four bits kept)
print(bin(b | 0b00000001))            # 0b10110011    (bit 0 turned on)
print(bin(b ^ 0b11111111))            # 0b1001101     (every bit flipped)
print(bin(b >> 4), bin(1 << 3))       # 0b1011 0b1000
```

`bin()` drops leading zeros, which is why `0b00001111` masked with 178 prints as `0b10`. The value is the same; only the display is shorter. Use `format(n, "08b")` when you want all eight digits.

A Trap: NOT in Python

In C, `~` on a byte stays within a byte. Python integers have no fixed width, so `~b` is defined as `-b - 1`, a negative number. To get the byte you meant, mask it back down:

```
print(~b)               # -179
print(~b & 0xFF)         # 77
```

This matters mostly when clearing a bit: `flags & ~MASK` works in Python without the extra `& 0xFF`, because a negative number behaves as if it had endless leading 1 bits and the AND with a small positive number gives the right small answer. It is only when you *store or print* a bare `~x` that the negative number surprises you.

Flags: Test, Set, Clear, Toggle

A **flags byte** gives each bit its own meaning. Name each bit as a one-bit mask, then four recipes cover everything:

```
READONLY, HIDDEN, COMPRESSED, ENCRYPTED = 1 << 0, 1 << 1, 1 << 2, 1 << 3

f = 0
f |= READONLY | COMPRESSED            # set two flags at once
print(bin(f), bool(f & HIDDEN), bool(f & COMPRESSED))
# 0b101 False True
f &= ~READONLY                        # clear one
print(bin(f))                         # 0b100
f ^= HIDDEN                           # toggle: on
print(bin(f))                         # 0b110
f ^= HIDDEN                           # toggle: off again
print(bin(f))                         # 0b100
```

GOAL	RECIPE	WHY IT WORKS
Test	`f & MASK`	Non-zero only if that bit is on
Set	`f | MASK`	OR can only turn bits on
Clear	`f & ~MASK`	The inverted mask keeps everything except that bit
Toggle	`f ^ MASK`	XOR with 1 flips, XOR with 0 leaves alone

Fields Wider Than One Bit

A bit field may be several bits wide: a 2-bit version number, a 4-bit length. To read one, **shift it down to bit 0, then mask to its width**. A mask of width `w` is `(1 << w) - 1`: width 2 gives `0b11`, width 4 gives `0b1111`. To write one, clear its old bits and OR in the new value shifted up:

```
def get_field(value, shift, width):
    return (value >> shift) & ((1 << width) - 1)

def set_field(value, shift, width, new):
    mask = ((1 << width) - 1) << shift
    if new >> width:
        raise ValueError("value does not fit")
    return (value & ~mask) | (new << shift)
```

```
v = 0b00101101
print(get_field(v, 4, 2), get_field(v, 0, 4))      # 2 13
w = set_field(v, 4, 2, 3)
print(bin(w), get_field(w, 4, 2))              # 0b111101 3
try:
    set_field(v, 4, 2, 4)
except ValueError as e:
    print(e)                                # value does not fit
```

The `new >> width` check refuses a value too big for the field. Without it, the extra bit would silently spill into the neighbouring field, which is the classic bug with packed data. Reading is forgiving, since the mask trims anything extra; writing needs the guard.

A Real Flags Byte: gzip

The gzip file format starts with a 10-byte header whose fourth byte, FLG, is a flags byte. Per the gzip specification (RFC 1952) (check): bit 0 is a text hint, bit 1 a header checksum, bit 2 an extra-data field, bit 3 a stored file name and bit 4 a comment; bits 5 to 7 are reserved and must be zero. Python's `gzip` module will make a sample for us in memory:

```
import gzip, io

buf = io.BytesIO()
with gzip.GzipFile(filename="notes.txt", fileobj=buf, mode="wb", mtime=0) as g:
    g.write(b"hello")
raw = buf.getvalue()
print(raw[:10].hex(" "))      # 1f 8b 08 08 00 00 00 00 02 ff
print(raw[3], bin(raw[3]))      # 8 0b1000
print(bool(raw[3] & 8), bool(raw[3] & 4))   # True False
```

`1f 8b` is the gzip magic number from Binary File Formats in Practice 4, `08` the compression method (deflate) and the next `08` is FLG: only bit 3 is on, so a file name follows the header. The last two bytes of the header (`02` and `ff`) are extra flags and an operating-system code that this chapter does not need (check what they mean in the specification before relying on them).

Naming Bits With IntFlag

Hand-written masks are error-prone. The standard library's `enum.IntFlag` gives every bit a name and still behaves as an integer, so it reads and writes straight into a struct:

```
import enum

class GzFlag(enum.IntFlag):
    TEXT = 1
    HCRC = 2
    EXTRA = 4
    NAME = 8
    COMMENT = 16

fl = GzFlag(raw[3])
print(int(fl), GzFlag.NAME in fl, GzFlag.EXTRA in fl)    # 8 True False
print(int(GzFlag.NAME | GzFlag.TEXT))                    # 9
```

The text shown when you print an `IntFlag` differs between Python versions (check), so this chapter prints plain integers. Note too that older versions of `IntFlag` accept unknown bits silently, while newer ones may raise or keep them depending on the class's boundary setting (check), so a format reader should still test the reserved bits itself.

A Packed 16-Bit Field: the DOS Date

Some formats pack a whole date into two bytes. The classic MS-DOS layout, still used by the ZIP format for file times (check), gives the year 7 bits, the month 4 and the day 5:

BITS	WIDTH	FIELD
15 to 9	7	Year, counted from 1980
8 to 5	4	Month, 1 to 12
4 to 0	5	Day, 1 to 31

```
import struct

dd = ((2026 - 1980) << 9) | (11 << 5) | 2
print(dd, hex(dd), bin(dd))                          # 23906 0x5d62 0b101110101100010
print((dd >> 9) + 1980, (dd >> 5) & 15, dd & 31)     # 2026 11 2
print(struct.pack("<H", dd).hex(" "))                # 62 5d
```

The whole date is one 16-bit integer, then stored little-endian like any other: bit packing decides the *value*, and the earlier chapters' byte-order rules decide how that value sits in the file. Reading is the reverse: `struct.unpack("<H", ...)` first, then shifts and masks. Because the year has only 7 bits, the format cannot represent a date after 2107 (1980 + 127).

IDEA	BYTE-SIZED FIELDS	BIT FIELDS
Smallest unit	1 byte, read with `struct`	1 bit, read with shifts and masks
Order question	Endianness (Binary File Formats in Practice 2)	Bit 0 is the least significant; fixed
Writing risk	Wrong size or sign	A value too big for its width spills into a neighbour
Validation	Check lengths	Check reserved bits are zero

Reserved Bits

Formats usually mark some bits "reserved, must be zero". A strict reader rejects a byte with those bits set; a lenient one ignores them. The choice has consequences: rejecting means an old reader refuses files from a newer writer that started using the bit, while ignoring means a corrupted byte can slip through unnoticed. This course's readers are strict, and say so in an error message that includes the offending bits.

Challenges

EXERCISE 1

Define four flags (READONLY, HIDDEN, COMPRESSED, ENCRYPTED) in bits 0 to 3 of a byte, with bits 4 to 7 reserved. Write `decode_flags(byte)` returning the list of names that are set, and `encode_flags(names)` returning the byte. Reject a byte with reserved bits set (including the bits in the message) and an unknown flag name. Show an encode and decode, the empty case, both failures, and prove that decoding then encoding returns the original for every value from 0 to 15.

View solution

EXERCISE 2

Write `pack_date` / `unpack_date` and `pack_time` / `unpack_time` for the DOS layouts (time: hour in bits 15 to 11, minute in bits 10 to 5, seconds divided by 2 in bits 4 to 0). Reject out-of-range values. Show 2026-11-02 and 14:30:45 packed as hex and as little-endian bytes, show that the seconds lose their odd digit on a round trip, show three rejections, and round-trip a spread of dates.

View solution

EXERCISE 3

Write `read_header(raw)` for the gzip header: check the magic number, the method (8) and the reserved bits of FLG, return the list of set flag names and the file name when the name flag is set (a zero-terminated string after the 10-byte header). Refuse the extra-data flag as not handled. Build files with `gzip.GzipFile` with and without a name, show both results, show five failures (magic, short, reserved, unterminated name, extra flag), and confirm `gzip.decompress` still works on the named file.

View solution

QUICK REFERENCE

- **Bit numbering:** bit 0 is the least significant (value 1); bit n has value `1 << n`
- **Test / set / clear / toggle:** `f & M`, `f | M`, `f & ~M`, `f ^ M`
- **Read a field:** `(v >> shift) & ((1 << width) - 1)`
- **Write a field:** clear with `v & ~mask`, then OR in `new << shift`; reject values that do not fit
- **NOT:** `~x` is negative in Python; mask with `& 0xFF` to get a byte
- **Multi-byte fields:** unpack the integer with `struct` first, then shift and mask
- **Reserved bits:** decide on purpose whether to reject or ignore them

- **IntFlag:** names for bits; print integers to avoid version differences

COMING NEXT

Binary File Formats in Practice 8 moves to records whose size is not fixed in advance: length prefixes, type-length-value layouts, and records nested inside records.

Binary File Formats in Practice

Chapter 8 · Length-Prefixed and TLV Records

Fixed layouts only go so far. A name, a comment or an embedded image has no size known in advance, so a format has to say how long each piece is. The standard answer is to write the length first and the data after it. Stacking that idea gives **type-length-value** (TLV) records, the shape behind a great many real formats, and letting a record's value hold more records gives nesting. This chapter builds all three and shows why a length is both the key to reading and the first thing that can go wrong. It uses the struct tools from Binary File Formats in Practice 3 and the byte-order rules from Binary File Formats in Practice 2.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed. Everything is done in memory, so nothing is left behind.

The Length Prefix

A **length-prefixed** field is a count followed by exactly that many bytes. The reader never has to search for an end marker; it reads the count, then takes that many bytes. Here a 2-byte little-endian length comes before a UTF-8 string:

```
import struct

def pack_str(s):
    data = s.encode("utf-8")
    return struct.pack("<H", len(data)) + data

def read_str(buf, pos):
    (n,) = struct.unpack_from("<H", buf, pos)
    pos += 2
    return buf[pos:pos + n].decode("utf-8"), pos + n

blob = pack_str("hello") + pack_str("café")
print(blob.hex(" "))
# 05 00 68 65 6c 6c 6f 05 00 63 61 66 c3 a9
s1, pos = read_str(blob, 0)
```

```
s2, pos = read_str(blob, pos)
print(s1, s2, pos, len(blob))               # hello café 14 14
print(len("café"), len("café".encode("utf-8")))   # 4 5
```

Two habits are worth noticing. The reader returns the *new position* along with the value, so the next read starts where this one ended; that is how you walk through a file of mixed fields. And the length counts **bytes, not characters**: "café" is four characters but five bytes, because the accented letter takes two bytes in UTF-8. Measure the encoded bytes, never the string.

How Big Should the Length Be?

The width of the length field is a design decision that limits the format forever:

LENGTH FIELD	LARGEST VALUE	COST
1 byte	255	Cheap, but short names or blobs overflow it
2 bytes	65,535	A common compromise
4 bytes	about 4.29 billion	Roomy, but 4 bytes on every tiny field
Variable (varint)	Effectively unlimited	Small values are small; slightly more code

A **varint** spends one byte on small numbers and more on big ones. The scheme used by several well-known formats (Protocol Buffers, DWARF and others use variants of it) (check) stores seven bits per byte, lowest bits first, and sets the top bit of a byte to mean "another byte follows":

```
def enc_varint(n):
    out = bytearray()
    while True:
        low = n & 0x7F
        n >>= 7
        if n:
            out.append(low | 0x80)
        else:
            out.append(low)
            return bytes(out)
```

```
def dec_varint(buf, pos):
    result = shift = 0
    while True:
        b = buf[pos]
        pos += 1
        result |= (b & 0x7F) << shift
        if not b & 0x80:
            return result, pos
        shift += 7

for n in (0, 1, 127, 128, 300, 16384):
    e = enc_varint(n)
    print(n, e.hex(" "), dec_varint(e, 0))
# 0 00 (0, 1)
# 1 01 (1, 1)
# 127 7f (127, 1)
# 128 80 01 (128, 2)
# 300 ac 02 (300, 2)
# 16384 80 80 01 (16384, 3)
```

Read 300 by hand: `ac 02` means the first byte contributes its low seven bits (`0x2c`, 44) and says more follows; the second contributes `2 << 7`, which is 256. Total 300. This reader has a flaw we return to below: it trusts the input to end. A stream of bytes that all have the top bit set would make it run until it falls off the data, and in a language without bounds checks, past it.

Type-Length-Value

A length prefix tells you how much; it does not tell you *what*. A **TLV** record adds a type tag in front: one byte saying what the record means, a length, then the value. A file becomes a sequence of such records. The header here is one type byte and a 2-byte little-endian length, so `struct.Struct("<BH")` describes it and its `size` is 3:

```
HEADER = struct.Struct("<BH")      # type, length

def pack_tlv(t, value):
    return HEADER.pack(t, len(value)) + value

def iter_tlv(buf):
```

```
    pos = 0
    while pos < len(buf):
        if len(buf) - pos < HEADER.size:
            raise ValueError(f"truncated header at offset {pos}")
        t, n = HEADER.unpack_from(buf, pos)
        start = pos + HEADER.size
        if start + n > len(buf):
            raise ValueError(f"record at {pos} wants {n} bytes, {len(buf) - start} left"
        yield t, buf[start:start + n]
        pos = start + n

data = pack_tlv(1, b"Ada") + pack_tlv(2, bytes([36])) + pack_tlv(99, b"??") + pack_tlv(1
print(data.hex(" "))
# 01 03 00 41 64 61 02 01 00 24 63 02 00 3f 3f 01 00 00
for t, v in iter_tlv(data):
    print(t, v)
# 1 b'Ada'
# 2 b'$'
# 99 b'??'
# 1 b''
```

The last record has length 0, which is perfectly valid: an empty value is still a record. Notice that the iterator never needs to know what any type *means*. It splits the bytes into records by length alone, and a second layer decides what each type is for.

Why TLV Ages Well: Skipping the Unknown

That separation is the point. Suppose a newer program starts writing type 99 and an older one has never heard of it. Because every record states its own length, the old reader can step over the unknown record and carry on with the rest:

```
KNOWN = {1: "name", 2: "age"}
for t, v in iter_tlv(data):
    print(KNOWN.get(t, f"unknown-{t}"), len(v))
# name 3
# age 1
# unknown-99 2
```

```
# name 0
```

Compare that with the fixed layouts of earlier chapters, where an unexpected extra field shifts everything after it and ruins the read.

IDEA	FIXED LAYOUT (STRUCT)	TLV RECORDS
Field positions	Known in advance	Found by walking the lengths
Random access	Jump straight to a field	Must scan from the start (or keep an index)
New field added	Breaks old readers	Old readers can skip it
Optional fields	Wasted space or flags	Just leave the record out
Overhead	None	Type and length on every record

Trusting the Length Is the Danger

The length is read from the file, so it is whatever the file says. The iterator above checks it against the bytes that remain before slicing. Slicing alone would not catch a lie, because Python quietly returns a shorter slice when the end is too close. Two kinds of damage show up:

```
for bad in (data[:14], data[:-1]):
    try:
        list(iter_tlv(bad))
    except ValueError as e:
        print(e)
# record at 10 wants 2 bytes, 1 left
# truncated header at offset 15
```

The first is a value cut short; the second is a header cut short. Both are reported with the offset, which makes a corrupt file far easier to diagnose than an anonymous failure. Binary File Formats in Practice 9 turns this into a full checklist for untrusted input.

Nested Records

If a record's value is itself a run of TLV records, you have a tree. Mark certain types as **containers**: when the reader meets one, it runs the same iterator over that record's value. The key point is that the inner reader is handed *only the value's bytes*, so a child can never read beyond its parent.

```
PERSON, NAME, AGE, TAG = 0x10, 1, 2, 3
CONTAINERS = {PERSON}

def parse(buf, depth=0):
    if depth > 8:
        raise ValueError("nested too deeply")
    out = []
    for t, v in iter_tlv(buf):
        if t in CONTAINERS:
            out.append((t, parse(v, depth + 1)))
        else:
            out.append((t, v))
    return out

person = pack_tlv(PERSON, pack_tlv(NAME, b"Ada") + pack_tlv(AGE, bytes([36])) + pack_tlv
doc = pack_tlv(PERSON, pack_tlv(NAME, b"Bo") + person) + pack_tlv(NAME, b"top")
print(doc.hex(" "))
# 10 16 00 01 02 00 42 6f 10 0e 00 01 03 00 41 64 61 02 01 00 24 03 01 00 78 01 03 00 74
print(parse(doc))
# [(16, [(1, b'Bo'), (16, [(1, b'Ada'), (2, b'$'), (3, b'x')])]), (1, b'top')]
```

Read the hex: the outer `10 16 00` is a person of 22 bytes; inside it comes a name, then another `10 0e 00` person of 14 bytes. Each length covers everything inside it, including the headers of its children, so lengths nest like brackets. The `depth` limit matters: a hostile file can nest records thousands deep, and unchecked recursion would exhaust the stack.

Writing Nested Data: Lengths Come Last

Writing has the opposite problem to reading. A container's length is only known once its children are built, so the simple approach is to build the value first and wrap it afterwards, as `pack_tlv` does. (A writer streaming straight to a file would write a placeholder, then `seek` back and patch it, using the file-position skills from Binary File Formats in Practice 1.)

Design Choices to Make on Purpose

- **Does the length include its own header?** Some formats count the whole record, others only the value. Here it counts only the value. Pick one and document it; mixing them causes off-by-header bugs.
- **Padding.** Some formats pad values to an even size, as the RIFF chunks in Binary File Formats in Practice 6 do. If so, the pad byte is not counted in the length.
- **Byte order.** Every multi-byte length needs a stated endianness; this chapter uses little-endian.
- **Duplicates and order.** Can a type appear twice? Does order matter? A reader should not have to guess.
- **Unknown types.** Skip them, keep them, or reject them. Skipping and keeping give the best compatibility; rejecting is stricter.

Challenges

EXERCISE 1

Write `pack_strings(items)` and `unpack_strings(buf)` for a list of strings: a varint count, then each string as a varint byte length followed by its UTF-8 bytes. The reader must bound the varint to five bytes, check every length against the bytes remaining, and reject trailing bytes. Show a round trip of a list containing an empty string, an accented string and a 128-character string, the size of the result and the first twelve bytes in hex, the prefix for 127 and 128, four failures (cut short, trailing byte, endless varint, empty input), and an unencodable string being refused.

View solution

EXERCISE 2

Write a TLV encoder and decoder for a record with a name (type 1, UTF-8), an age (type 2, one byte) and a score (type 3, little-endian 32-bit float), using a 1-byte type and a 2-byte length. The decoder must keep unknown records so that re-encoding gives the same bytes back. Show the encoded hex, a file with an unknown record added, that re-encoding it matches, that the record order does not matter, and three failures (wrong value size, cut value, cut header). The encoder must refuse a value longer than the length field allows.

View solution

EXERCISE 3

Model a small directory tree as nested TLV: a folder record holds its name and its children; a file record holds a name and a 4-byte size. Write `build(tree)` and `parse(buf)`, a function that prints the tree indented, and prove that parsing a built tree returns the original. Show a nesting-depth failure, a child whose length overruns its parent, and a truncated file.

View solution

QUICK REFERENCE

- **Length prefix:** count first, then exactly that many bytes; the count is in bytes, not characters
- **Walking a buffer:** read a value, return the new position, carry on from there
- **Varint:** 7 bits per byte, low bits first, top bit set means more follows; bound its length when reading
- **TLV:** type, length, value; a file is a run of records; `struct.Struct("<BH")` for a 1-byte type and 2-byte length
- **Forward compatibility:** skip or keep unknown types using their length
- **Check before slicing:** compare every length with the bytes that remain; report the offset in the error
- **Nesting:** run the iterator on a container's value only, and limit the depth
- **Writing:** build the value, then wrap it with its length

COMING NEXT

Binary File Formats in Practice 9 collects the defences this chapter began: truncation, lying lengths, size limits and other ways to handle untrusted input without crashing or hanging.

Binary File Formats in Practice

Chapter 9 · Robust Parsing

Every parser in this course so far has been fed files it wrote itself. Real files are not so polite. They get cut short by a failed download, damaged by a bad disk, written by buggy programs, or built on purpose to break the reader. A parser that is correct for good input and unpredictable for bad input is only half finished. This chapter collects the defences that Binary File Formats in Practice 8 began: a single checked gate for reading, limits on counts and sizes, protection from decompression bombs and pointer loops, and a small fuzzer that proves the parser fails cleanly.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed. Everything is done in memory, so nothing is left behind. The snippets build on one another: run them in order in one session.

Every Number in the File Is a Claim

The core idea is simple: **nothing read from a file is known to be true**. A length, a count, an offset, a version, even a "this record is 12 bytes" are only statements the file makes about itself. Robust parsing means checking each statement against what you can actually see before acting on it. The failures fall into a few families:

PROBLEM	EXAMPLE	DEFENCE
Truncation	The file ends in the middle of a field	Check bytes remaining before every read
Lying length	A record claims more bytes than the file has	Compare the length with the bytes that remain
Lying count	"4 billion items" in a 20-byte file	A fixed limit, and a check that the items could fit
Expansion	A tiny compressed stream that inflates to gigabytes	Cap the output size while decompressing

PROBLEM	EXAMPLE	DEFENCE
Loops and depth	An offset that points back at itself; records nested thousands deep	Visited set, hop limit, depth limit
Bad content	Invalid UTF-8, an unknown version	Validate and reject with a clear message
Leftovers	Extra bytes after the last record	Decide: reject, or skip on purpose

One Gate for Every Read

The easiest way to forget a check is to scatter reads through the code. Instead, put them behind one small class whose only job is to hand out bytes safely, and make it raise one error type for every kind of bad input. Callers then catch a single `FormatError` rather than a mixture of whatever Python happens to raise.

```
import struct, zlib

class FormatError(ValueError):
    """The input is not a valid file. Raised for every kind of bad input."""

class Reader:
    def __init__(self, buf):
        self.buf = bytes(buf)
        self.pos = 0

    def remaining(self):
        return len(self.buf) - self.pos

    def take(self, n):
        if n < 0 or n > self.remaining():
            raise FormatError(f"need {n} bytes at offset {self.pos}, {self.remaining()}
        chunk = self.buf[self.pos:self.pos + n]
        self.pos += n
        return chunk

    def unpack(self, fmt):
        st = struct.Struct(fmt)
        return st.unpack(self.take(st.size))
```

```
r = Reader(b"\x01\x02\x03")
print(r.unpack("<H"), r.remaining())              # (513,) 1
try:
    r.unpack("<I")
except FormatError as e:
    print(e)                                     # need 4 bytes at offset 2, 1 left
try:
    r.take(-1)
except FormatError as e:
    print(e)                                     # need -1 bytes at offset 2, 1 left
print(b"abc"[5:9], b"abc"[5:9] == b"")      # b'' True
try:
    b"abc"[5]
except IndexError as e:
    print("IndexError:", e)               # IndexError: index out of range
```

Three details earn their place here. The check is `n > remaining` *before* slicing, because Python slicing never complains: asking for bytes past the end quietly returns a shorter, even empty, result, and the parser carries on with garbage. A negative request is refused too, since `buf[pos:pos + n]` with a negative `n` would walk backwards. And the message carries the offset and the shortfall, which turns "invalid file" into something you can debug with a hex dump from Binary File Formats in Practice 1.

Indexing is the opposite trap: `buf[5]` past the end raises `IndexError`, which has nothing to do with your format and would escape a caller expecting `FormatError`. Routing everything through `take` avoids both.

Counts and Sizes Are Claims Too

A count field is the classic way to make a program allocate far more than the file justifies. Suppose a format starts with a 4-byte record count, and a hostile file says four billion. A parser that builds a list of that size, or loops that many times before noticing the data has run out, wastes huge amounts of memory or time. Two cheap checks stop it. First, a hard limit chosen by you. Second, a feasibility test: if each record needs at least a minimum number of bytes, the count cannot exceed `remaining // minimum`.

```
# u32 count, then count x (u16 id, u8 name length, name bytes)
```

```
MIN_RECORD = 3
MAX_RECORDS = 1000

def parse_list(buf):
    r = Reader(buf)
    (count,) = r.unpack("<I")
    if count > MAX_RECORDS:
        raise FormatError(f"count {count} over limit {MAX_RECORDS}")
    if count * MIN_RECORD > r.remaining():
        raise FormatError(f"count {count} cannot fit in {r.remaining()} bytes")
    out = []
    for _ in range(count):
        ident, n = r.unpack("<HB")
        try:
            name = r.take(n).decode("utf-8")
        except UnicodeDecodeError as e:
            raise FormatError(f"bad UTF-8 at offset {r.pos - n}") from e
        out.append((ident, name))
    return out

good = struct.pack("<I", 2) + struct.pack("<HB", 7, 3) + b"Ada" + struct.pack("<HB", 8, 
print(parse_list(good))                       # [(7, 'Ada'), (8, 'Bo')]
for label, bad in [
    ("huge count", struct.pack("<I", 4_000_000_000) + b"xx"),
    ("count too big for data", struct.pack("<I", 500) + good[4:]),
    ("cut name", good[:-1]),
    ("bad utf-8", struct.pack("<I", 1) + struct.pack("<HB", 1, 2) + b"\xff\xfe"),
]:
    try:
        parse_list(bad)
    except FormatError as e:
        print(label, "->", e)
# huge count -> count 4000000000 over limit 1000
# count too big for data -> count 500 cannot fit in 11 bytes
# cut name -> need 2 bytes at offset 13, 1 left
# bad utf-8 -> bad UTF-8 at offset 7
```

Notice that the same check appears in different clothes everywhere: a length against the bytes left, a count against the bytes left, a decode failure turned into the shared error type. The last one uses `raise ... from e` so the original exception is kept as the cause while callers still catch only `FormatError`.

Choosing Limits

A limit is a promise about the largest thing your program will accept. Pick them on purpose and write them down next to the format:

- **Total file size.** If you read the whole file into memory, refuse files over a size you can afford before reading them (check `os.path.getsize` or read at most the limit plus one byte).
- **Item counts and string lengths.** Far below what the length field could express. A 4-byte length can say four gigabytes; a name field rarely needs more than a few hundred bytes.
- **Nesting depth.** A small number, such as the 8 used in Binary File Formats in Practice 8. Python's default recursion limit is 1000 frames (check), so a hostile file nested deeper than that would otherwise crash the parser with a `RecursionError` rather than a clean rejection.
- **Total work.** Loops over records should have an upper bound that does not depend only on the file's own claims.
- **Decompressed size.** See the next section.

Decompression Bombs

Compression makes a particularly nasty lie possible, because the input is tiny but the output is not. A run of zeros compresses extremely well: fifty million zero bytes squeeze into about 48 thousand, a ratio above a thousand to one. A parser that calls `zlib.decompress(data)` on untrusted input will happily build the full fifty million bytes (and a determined attacker can do far worse). The fix is to decompress with a **cap** using a decompressor object, which accepts a `max_length` and stops there:

```
bomb = zlib.compress(b"\x00" * 50_000_000, 9)
print(len(bomb))                              # 48610

def inflate(data, max_out):
    d = zlib.decompressobj()
    try:
        out = d.decompress(data, max_out + 1)
    except zlib.error as e:
        raise FormatError(f"corrupt stream: {e}") from e
    if len(out) > max_out:
        raise FormatError(f"output over {max_out} bytes")
```

```
    if not d.eof:
        raise FormatError("stream is cut short")
    return out

print(len(inflate(zlib.compress(b"hello" * 100), 1000)))   # 500
for label, data in [("bomb", bomb), ("cut", zlib.compress(b"hello" * 100)[:-4])]:
    try:
        inflate(data, 1_000_000)
    except FormatError as e:
        print(label, "->", e)
# bomb -> output over 1000000 bytes
# cut -> stream is cut short
```

Asking for `max_out + 1` bytes is the trick: if more than `max_out` come back, the stream wanted to produce more than you allow, and you can refuse without ever having held the whole thing. The `eof` flag then separates a stream that genuinely ended from one that was cut off. The same care applies to the PNG chunk data from Binary File Formats in Practice 5, whose image data is compressed: a decoder that trusts the header's width and height to size its buffers inherits the same problem.

Offsets, Loops and Pointer Chains

Some formats do not store records one after another; they store **offsets** that say where the next thing lives. An offset is another claim, with a new way to fail: it can point outside the file, or back at something already visited, which makes a loop. A parser that follows "next" pointers until it sees an end marker will spin forever on a file whose chain has no end.

```
def parse_chain(buf, max_hops=16):
    # each node: u8 value, u16 offset of the next node (0xFFFF = end)
    seen, pos, values = set(), 0, []
    while pos != 0xFFFF:
        if pos in seen:
            raise FormatError(f"loop: offset {pos} visited twice")
        if len(seen) >= max_hops:
            raise FormatError("chain too long")
        seen.add(pos)
        if pos + 3 > len(buf):
            raise FormatError(f"node at {pos} is outside the data")
```

```
            v, nxt = struct.unpack_from("<BH", buf, pos)
            values.append(v)
            pos = nxt
        return values

    ok = bytes([10, 3, 0]) + bytes([20, 0xFF, 0xFF])
    loop = bytes([10, 3, 0]) + bytes([20, 0, 0])
    far = bytes([10, 99, 0])
    print(parse_chain(ok))                          # [10, 20]
    for label, b in [("loop", loop), ("far", far)]:
        try:
            parse_chain(b)
        except FormatError as e:
            print(label, "->", e)
    # loop -> loop: offset 0 visited twice
    # far -> node at 99 is outside the data
```

The visited set catches a true loop at once; the hop limit is a second line of defence that also bounds a long but finite chain. Both are cheap, and a loop-free format that "never" produces cycles still deserves them, because the reader cannot know the writer was honest.

Checksums Detect Accidents, Not Attackers

A CRC, like the one on each PNG chunk in Binary File Formats in Practice 5, is excellent at noticing accidental damage: a flipped bit, a truncated transfer. It does nothing against someone who changes the data and recomputes the checksum, because a CRC has no secret.

```
    payload = b"hello world"
    framed = payload + struct.pack("<I", zlib.crc32(payload))

    def check(framed):
        body, (crc,) = framed[:-4], struct.unpack("<I", framed[-4:])
        if zlib.crc32(body) != crc:
            raise FormatError("checksum mismatch")
        return body

    print(check(framed))                            # b'hello world'
    flipped = bytes([framed[0] ^ 1]) + framed[1:]
    try:
```

```
    check(flipped)
except FormatError as e:
    print(e)                                  # checksum mismatch
forged = flipped[:-4] + struct.pack("<I", zlib.crc32(flipped[:-4]))
print(check(forged))                          # b'iello world'
```

So a checksum is a good *first* filter for corruption, but it never replaces the structural checks above: a file with a valid CRC can still hold a lying length. Protection against deliberate tampering needs a keyed construction such as a message authentication code or a digital signature, which is outside this course.

Prove It: Break Your Own Files

Careful code is a belief; a test is evidence. The cheapest useful test for a parser is a small **fuzzer**: take a valid file, damage it in every simple way, and check that the parser either accepts the result or rejects it with `FormatError`. Anything else, whether `struct.error`, `IndexError`, `RecursionError` or a hang, is a bug. The simplest damage is to cut the file at every possible length and to overwrite each byte with a few extreme values:

```
def parse_tlv(buf, depth=0):
    if depth > 8:
        raise FormatError("nested too deeply")
    r = Reader(buf)
    out = []
    while r.remaining():
        t, n = r.unpack("<BH")
        v = r.take(n)
        out.append((t, parse_tlv(v, depth + 1) if t == 0x10 else v))
    return out

def tlv(t, v):
    return struct.pack("<BH", t, len(v)) + v

doc = tlv(0x10, tlv(1, b"Bo") + tlv(0x10, tlv(1, b"Ada") + tlv(2, b"\x24"))) + tlv(1, b"
print(len(doc), parse_tlv(doc))
# 27 [(16, [(1, b'Bo'), (16, [(1, b'Ada'), (2, b'$')])]), (1, b'top')]

def fuzz(parser, doc):
```

```
    results = {"ok": 0, "rejected": 0}
    crashes = []
    cases = [doc[:i] for i in range(len(doc))]
    for i in range(len(doc)):
        for val in (0x00, 0xFF, doc[i] ^ 0x80):
            cases.append(doc[:i] + bytes([val]) + doc[i + 1:])
    for case in cases:
        try:
            parser(case)
            results["ok"] += 1
        except FormatError:
            results["rejected"] += 1
        except Exception as e:
            crashes.append((type(e).__name__, case))
    return len(cases), results, crashes

n, res, crashes = fuzz(parse_tlv, doc)
print(n, res, len(crashes))                  # 108 {'ok': 55, 'rejected': 53} 0
```

Of the 108 damaged files, 55 still parse (a changed value byte is still a valid value, and some cuts land exactly between records) and 53 are rejected cleanly. None crash. Now the same test on a sloppy parser that reads lengths without checking them:

```
def sloppy(buf):
    pos, out = 0, []
    while pos < len(buf):
        t = buf[pos]
        (n,) = struct.unpack_from("<H", buf, pos + 1)
        out.append((t, buf[pos + 3:pos + 3 + n]))
        pos += 3 + n
    return out

n, res, crashes = fuzz(sloppy, doc)
print(n, res, len(crashes), sorted({c[0] for c in crashes}))
# 108 {'ok': 104, 'rejected': 0} 4 ['error']
```

The sloppy parser has two failures, and the second is the worse. Four cases crash with `struct.error` (a header cut short). But it *accepts* 104 of 108, including many it should have

refused: a record whose length overruns the data silently becomes a shorter record, so a damaged file produces plausible but wrong results with no warning. Silent acceptance of bad data is more dangerous than a crash, because nobody notices.

> **WHERE THE STAKES ARE HIGHER**
>
> In Python the cost of these mistakes is usually an exception or wasted memory. In languages without bounds checking, such as C, a lying length can read or write beyond a buffer, and that is a well-known source of serious security flaws (check). The habits are the same in every language; only the consequences differ.

A Robust Parser Checklist

- **One gate.** All reads go through a helper that checks bytes remaining. No raw indexing of untrusted lengths.
- **One error type.** Convert `struct.error`, `UnicodeDecodeError` and `zlib.error` into `FormatError` so callers handle one thing.
- **Check before you allocate or loop.** Counts and sizes against a limit and against what could fit.
- **Cap expansion.** Decompress with an output limit, and compare the declared size with the real one.
- **Bound the structure.** Depth limit, hop limit, visited set.
- **Validate content.** Magic, version, text encoding, reserved fields that must be zero.
- **Decide about leftovers.** Trailing bytes are either an error or an intentional extension point; never an accident.
- **Report the offset.** Every error message says where in the file it happened.
- **Fuzz it.** Truncate and mutate a good file, and require that only `FormatError` comes out.

Challenges

EXERCISE 1

Write a parser for a small inventory file: the magic bytes `INV1`, a 1-byte version (only 1 is accepted), a 2-byte little-endian item count, then that many items, each a 1-byte name length, the UTF-8 name

(never empty), a 4-byte quantity and a 2-byte price. Build it on a checked reader. Enforce a file-size limit of 65,536 bytes, a limit of 100 items, a check that the count could fit in the bytes left, and a refusal of trailing bytes. Show a good file parsed, an empty list, and eleven bad inputs each rejected with its own clear message.

View solution

EXERCISE 2

Write a safe loader for a blob made of a 4-byte little-endian declared size followed by a zlib stream. Refuse a declared size over 1,000,000 before decompressing, cap the decompressor's output, and check the real size equals the declared size. Show a good blob, a 50-million-byte bomb (with an honest size and with a lying one), a size that is too small, too large and absurd, and a cut stream, trailing bytes, corrupt data, non-zlib data and a blob too short for its size field. Every failure must be a `FormatError`.

View solution

EXERCISE 3

Build a fuzz harness that cuts a seed file at every length and replaces each byte with 0x00, 0xFF and the byte with its top bit flipped (skipping replacements equal to the original), counting accepted, rejected and crashed cases and recording the first case for each crash type. Run it against a careful nested type-length-value parser and a sloppy one, and finish by feeding both a record nested 50 levels deep.

View solution

QUICK REFERENCE

- **Principle:** every length, count, offset and size in a file is a claim to verify
- **Slicing never fails:** `buf[a:b]` past the end returns less; indexing raises `IndexError`; check first
- **Checked reader:** `take(n)` refuses `n < 0` or `n > remaining` and names the offset
- **One error type:** wrap `struct.error`, `UnicodeDecodeError`, `zlib.error` with `raise FormatError(...) from e`
- **Counts:** hard limit, plus `count * min_size <= remaining`
- **Bombs:** `zlib.decompressobj().decompress(data, max_out + 1)`, then check length and `eof`
- **Loops and depth:** visited set, hop limit, recursion depth limit

- **CRC:** catches accidents, not tampering
- **Fuzz:** truncate at every length, mutate every byte; only `FormatError` is acceptable

COMING NEXT

Binary File Formats in Practice 10 is the capstone: you design a tiny container format of your own, write the reader and writer, apply the defences from this chapter, and test it with round trips and a fuzzer.

Binary File Formats in Practice

Chapter 10 · Capstone: A Tiny Container Format

Nine chapters of taking other people's formats apart end here with building one. A **container format** is a file that holds other files: think of any archive you have opened. Yours will be small enough to read in one sitting, yet it will use every idea in the course: a magic number and version, a fixed header read with `struct`, length-prefixed records, a flags field, a checksum, optional compression, and the defensive reader from Binary File Formats in Practice 9. The finished program is about a hundred lines, and you will test it with round trips, hand-damaged files, a decompression bomb and a fuzzer.

> **WHAT WAS RUN**
>
> Every snippet and solution in this chapter was executed, and the output shown in the comments is what Python printed. Everything is done in memory (apart from one exercise that uses a temporary folder which is deleted afterwards). The snippets build on one another: run them in order in one session. The format is invented for this course, so it is not compatible with any real tool.

Step 1: Design Before You Code

Writing a format down before writing any code is the habit that separates a format from an accident. The decisions are few, and each one answers a question from an earlier chapter. We call the format **TBX** (tiny box), version 1. All integers are little-endian, as in Binary File Formats in Practice 2.

PART	FIELDS	STRUCT FORMAT
Header (8 bytes)	magic `TBX1`, version, flags, entry count	`<4sBBH`
Entry header (13 bytes)	name length, stored size, original size, CRC-32 of the original data	`<BIII`
Entry body	the name in UTF-8, then the stored data	raw bytes

```
TBX file
```

```
├── header          8 bytes     magic "TBX1", version 1, flags, count
└── entry x count
    ├── entry header   13 bytes     name_len, stored, original, crc32
    ├── name           name_len bytes   UTF-8, no "/" or "\", not empty
    └── data           stored bytes     raw, or zlib if flag bit 0 is set
```

The design choices, and why:

- **Magic and version up front.** The reader can reject a stranger's file in the first eight bytes (Binary File Formats in Practice 4). The version lets a later format change shape without confusing old readers.
- **Flags with a rule for unknown bits.** Bit 0 means "data is zlib-compressed". A reader that meets any other bit set must refuse, because it does not know what that bit promises (Binary File Formats in Practice 7).
- **Stored size and original size.** Both are written so that the reader can cap decompression at the declared original size, and so it can verify the claim afterwards.
- **CRC of the original data.** It catches accidental damage after decompression. Remember from Binary File Formats in Practice 9 that it detects accidents, not tampering.
- **Counts instead of an end marker.** The entry count in the header lets the reader reject an impossible count before looping.
- **Limits written next to the format.** At most 1,000,000 bytes per file, 100 entries and 500,000 bytes per entry, with names of 1 to 255 bytes. These are choices, not laws; the point is that they exist.

Step 2: The Writer

The writer is the easy half, but it still validates: a writer that can produce a file its own reader rejects is a bug. Names are checked by one shared function, used by both halves, so the rule lives in one place.

```
import struct, zlib

MAGIC = b"TBX1"
VERSION = 1
HEADER = struct.Struct("<4sBBH")    # magic, version, flags, entry count
```

```
ENTRY = struct.Struct("<BIII")      # name length, stored size, original size, crc32
FLAG_ZLIB = 0x01
KNOWN_FLAGS = FLAG_ZLIB

MAX_FILE = 1_000_000
MAX_ENTRIES = 100
MAX_ORIGINAL = 500_000

class FormatError(ValueError):
    """The input is not a valid TBX file, or cannot be written as one."""

def check_name(name):
    if not name or "/" in name or "\\" in name or name in (".", ".."):
        raise FormatError(f"bad entry name {name!r}")
    if len(name.encode("utf-8")) > 255:
        raise FormatError("entry name too long")

def write_tbx(entries, compress=False):
    if len(entries) > MAX_ENTRIES:
        raise FormatError("too many entries")
    seen, body = set(), []
    for name, data in entries:
        check_name(name)
        if name in seen:
            raise FormatError(f"duplicate name {name!r}")
        seen.add(name)
        if len(data) > MAX_ORIGINAL:
            raise FormatError(f"{name!r} is over {MAX_ORIGINAL} bytes")
        stored = zlib.compress(data, 9) if compress else data
        raw_name = name.encode("utf-8")
        body.append(ENTRY.pack(len(raw_name), len(stored), len(data), zlib.crc32(data)))
        body.append(raw_name)
        body.append(stored)
    flags = FLAG_ZLIB if compress else 0
    return HEADER.pack(MAGIC, VERSION, flags, len(entries)) + b"".join(body)

files = [("hello.txt", b"Hello, world!\n"), ("zeros.bin", bytes(300)), ("empty", b"")]
blob = write_tbx(files)
print(len(blob), blob[:8].hex(" "))
small = write_tbx(files, compress=True)
print(len(small), small[:8].hex(" "))

# 384 54 42 58 31 01 00 03 00
```

```
# 113 54 42 58 31 01 01 03 00
```

The uncompressed file is 384 bytes: the 8-byte header, three 13-byte entry headers, the names (9 + 9 + 5), and 14 + 300 + 0 bytes of data. The compressed one is 113 bytes because the 300 zero bytes shrink to almost nothing. Byte 4 is the version (`01`), and byte 5 is the flags: `00` in the first file, `01` in the second.

Step 3: The Reader

The reader is where the chapter on robust parsing pays off. It is built on the checked `Reader` class, converts every kind of failure into `FormatError`, and checks each claim before acting on it: file size, magic, version, flag bits, entry count (against a limit and against the bytes that could hold it), name encoding and content, duplicate names, declared sizes, the real decompressed size, the checksum, and finally the trailing bytes.

```
class Reader:
    def __init__(self, buf):
        self.buf = bytes(buf)
        self.pos = 0

    def remaining(self):
        return len(self.buf) - self.pos

    def take(self, n):
        if n < 0 or n > self.remaining():
            raise FormatError(f"need {n} bytes at offset {self.pos}, {self.remaining()}
        chunk = self.buf[self.pos:self.pos + n]
        self.pos += n
        return chunk

    def unpack(self, st):
        return st.unpack(self.take(st.size))

def inflate(data, expected):
    d = zlib.decompressobj()
    try:
        out = d.decompress(data, expected + 1)
    except zlib.error as e:
```

```
            raise FormatError(f"corrupt zlib data: {e}") from e
        if len(out) != expected or not d.eof or d.unused_data:
            raise FormatError(f"zlib data does not match the declared size {expected}")
        return out

    def read_tbx(buf):
        if len(buf) > MAX_FILE:
            raise FormatError(f"file over {MAX_FILE} bytes")
        r = Reader(buf)
        magic, version, flags, count = r.unpack(HEADER)
        if magic != MAGIC:
            raise FormatError(f"bad magic {magic!r}")
        if version != VERSION:
            raise FormatError(f"unsupported version {version}")
        if flags & ~KNOWN_FLAGS:
            raise FormatError(f"unknown flag bits {flags & ~KNOWN_FLAGS:#04x}")
        if count > MAX_ENTRIES:
            raise FormatError(f"entry count {count} over limit {MAX_ENTRIES}")
        if count * (ENTRY.size + 1) > r.remaining():
            raise FormatError(f"entry count {count} cannot fit in {r.remaining()} bytes")
        out, seen = [], set()
        for _ in range(count):
            start = r.pos
            name_len, stored, original, crc = r.unpack(ENTRY)
            try:
                name = r.take(name_len).decode("utf-8")
            except UnicodeDecodeError as e:
                raise FormatError(f"entry at {start}: name is not UTF-8") from e
            check_name(name)
            if name in seen:
                raise FormatError(f"duplicate name {name!r}")
            seen.add(name)
            if original > MAX_ORIGINAL:
                raise FormatError(f"{name!r}: declared size {original} over limit")
            data = r.take(stored)
            if flags & FLAG_ZLIB:
                data = inflate(data, original)
            elif stored != original:
                raise FormatError(f"{name!r}: stored size {stored} != original size {origina
            if zlib.crc32(data) != crc:
                raise FormatError(f"{name!r}: checksum mismatch")
            out.append((name, data))
        if r.remaining():
            raise FormatError(f"{r.remaining()} trailing bytes at offset {r.pos}")
        return out
```

```
print(read_tbx(blob) == files, read_tbx(small) == files)
print(read_tbx(write_tbx([])))
for name, data in read_tbx(small):
    print(f"{name:10} {len(data):4} bytes")

# True True
# []
# hello.txt    14 bytes
# zeros.bin   300 bytes
# empty         0 bytes
```

Read `read_tbx` once top to bottom and notice that it is a list of questions, in an order that never lets a later step trust something an earlier step has not checked. In `inflate`, requesting `expected + 1` bytes lets a single call prove that the stream would produce more than promised, without holding the excess; `unused_data` catches bytes left after the stream ends. The empty archive is legal and round-trips to an empty list.

Step 4: Round Trips

The first test of any reader and writer pair is the **round trip**: write something, read it back, and expect the same thing. Doing it with random contents, of random sizes, both compressed and not, catches the cases you did not think of. The random generator is seeded so the test gives the same answer every time.

```
import os, random

def roundtrip(entries, compress):
    return read_tbx(write_tbx(entries, compress)) == entries

random.seed(27)
trials = 0
for _ in range(200):
    entries = [(f"f{i}.dat", os.urandom(random.randint(0, 200)) if random.random() < 0.5
                else bytes(random.randint(0, 2000)))
               for i in range(random.randint(0, 8))]
    for compress in (False, True):
        assert roundtrip(entries, compress)
```

```
        trials += 1
print(trials, "round trips ok")
print(roundtrip([("é.txt", "héllo".encode())], True))

# 400 round trips ok
# True
```

The accented name shows that names travel as UTF-8, and that the length field counts bytes rather than characters: `é` is one character but two bytes.

Step 5: Hand-Made Bad Files

A round trip only proves the happy path. Now damage a known-good file on purpose, one field at a time, and read the error message: each one should name the problem and, where it can, the offset. `patched` overwrites a field at a known position, using the layout from the design table (the first entry header starts at offset 8).

```
def bad(label, buf):
    try:
        read_tbx(buf)
        print(label, "-> ACCEPTED")
    except FormatError as e:
        print(label, "->", e)

def patched(buf, offset, fmt, value):
    return buf[:offset] + struct.pack(fmt, value) + buf[offset + struct.calcsize(fmt):]

bad("empty file", b"")
bad("bad magic", b"XXXX" + blob[4:])
bad("version 2", patched(blob, 4, "<B", 2))
bad("unknown flag", patched(blob, 5, "<B", 0x82))
bad("huge count", patched(blob, 6, "<H", 60000))
bad("count too big for data", patched(blob, 6, "<H", 90))
bad("cut in the middle", blob[:-5])
bad("trailing bytes", blob + b"\x00")
bad("flipped data bit", blob[:30] + bytes([blob[30] ^ 1]) + blob[31:])
bad("size mismatch", patched(blob, 13, "<I", 15))
one = ENTRY.pack(1, 1, 1, zlib.crc32(b"x")) + b"ax"
bad("duplicate names", HEADER.pack(MAGIC, VERSION, 0, 2) + one + one)
```

```
bad("zlib cut", small[:-3])

# empty file -> need 8 bytes at offset 0, 0 left
# bad magic -> bad magic b'XXXX'
# version 2 -> unsupported version 2
# unknown flag -> unknown flag bits 0x82
# huge count -> entry count 60000 over limit 100
# count too big for data -> entry count 90 cannot fit in 376 bytes
# cut in the middle -> need 5 bytes at offset 379, 0 left
# trailing bytes -> 1 trailing bytes at offset 384
# flipped data bit -> 'hello.txt': checksum mismatch
# size mismatch -> 'hello.txt': stored size 14 != original size 15
# duplicate names -> duplicate name 'a'
# zlib cut -> need 8 bytes at offset 105, 5 left
```

Twelve different damages, twelve different messages, and no exception that is not a `FormatError`. Writing these cases is also the quickest way to find a missing check: each time you can imagine a field lying, add a line here and see what the reader says.

Step 6: A Bomb and a Fuzzer

The last two tests are the heavy ones. First a decompression bomb: a 48 KB stream that inflates to 50 million bytes, hidden in an entry that claims a modest size. Then the fuzzer from the previous chapter, with one small change: it records a label for each case, so a crash could be traced to the exact damage that caused it.

```
bomb = zlib.compress(bytes(50_000_000), 9)
print(len(bomb), "byte bomb")
liar = HEADER.pack(MAGIC, VERSION, FLAG_ZLIB, 1) + ENTRY.pack(1, len(bomb), 1000, 0) + b
bad("bomb, small claim", liar)
huge = HEADER.pack(MAGIC, VERSION, FLAG_ZLIB, 1) + ENTRY.pack(1, len(bomb), 50_000_000, 
bad("bomb, honest claim", huge)

# 48610 byte bomb
# bomb, small claim -> zlib data does not match the declared size 1000
# bomb, honest claim -> 'b': declared size 50000000 over limit
```

The first bomb lies about its size and is stopped by the capped decompressor; the second tells the truth and is stopped by the per-entry limit before any decompression starts. (The compressed size, 48,610 bytes, is what this machine's zlib produced and may differ slightly with another version (check).)

```
def fuzz(parser, seed):
    cases = [("cut at %d" % i, seed[:i]) for i in range(len(seed))]
    for i in range(len(seed)):
        for val in (0x00, 0xFF, seed[i] ^ 0x80):
            if val != seed[i]:
                cases.append((f"byte {i} -> {val:#04x}", seed[:i] + bytes([val]) + seed[
    ok = rejected = 0
    crashes = []
    for label, case in cases:
        try:
            parser(case)
            ok += 1
        except FormatError:
            rejected += 1
        except Exception as e:
            crashes.append((label, type(e).__name__))
    return len(cases), ok, rejected, crashes

for name, seed in (("plain", blob), ("zlib", small)):
    total, ok, rejected, crashes = fuzz(read_tbx, seed)
    print(name, len(seed), "bytes:", total, "cases,", ok, "accepted,", rejected, "reject

# plain 384 bytes: 1212 cases, 23 accepted, 1189 rejected, 0 crashes
# zlib 113 bytes: 422 cases, 27 accepted, 395 rejected, 0 crashes
```

Zero crashes in 1,634 damaged files: every one is either refused with `FormatError` or accepted. The accepted ones deserve a look, because accepting damaged input is what the fuzzer is really hunting. Here all 23 accepted cases in the uncompressed file turn out to be changes to a byte of an entry *name* (offsets 21 to 29, 57 to 65 and 379 to 383), which produce a different but still valid name. That is a real finding about the design, not a bug in the code: the CRC covers the data, and nothing covers the names. Exercise 2 closes that gap.

DESIGN LESSON

> A fuzzer tests your parser against your own idea of "valid". When it reports something accepted, ask whether the *format* should have noticed. Often the answer is a field nobody is protecting.

What You Would Add Next

A real format keeps growing, and the design above leaves room to do it safely:

- **New flags.** A reader that knows only bit 0 refuses a file using bit 1, instead of misreading it. Old files stay valid.
- **A new version.** Change the version byte when the layout changes, as Exercise 2 does with a whole-file checksum. A reader supports the versions it knows and rejects the rest by name.
- **An index.** Reading one file currently means walking every entry. A table of offsets at the end (like many real archives, which put the index at the end so files can be appended) allows direct access, at the price of the offset checks from Binary File Formats in Practice 9.
- **Tagged extra data.** A type-length-value area per entry, as in Binary File Formats in Practice 8, so that new metadata (timestamps, comments) can be added and unknown tags skipped.
- **Streaming.** The reader here takes the whole file as `bytes`. For huge archives you would read from a file object, keeping the same checks on a bounded buffer.

The Whole Course in One Table

Each chapter contributed something to the format you have just built:

CHAPTER	USED IN TBX AS
Binary File Formats in Practice 1	The hex dump (`hex(" ")`) used to look at the output, and binary reading and writing
Binary File Formats in Practice 2	Little-endian integers throughout
Binary File Formats in Practice 3	The two `struct.Struct` objects and their sizes
Binary File Formats in Practice 4	The magic number and version at the start

CHAPTER	USED IN TBX AS
Binary File Formats in Practice 5	CRC-32 with zlib, and compressed data
Binary File Formats in Practice 6	Fixed header followed by sized chunks of data
Binary File Formats in Practice 7	The flags byte and its unknown-bits rule
Binary File Formats in Practice 8	Length-prefixed names and data
Binary File Formats in Practice 9	The checked reader, limits, bomb protection, fuzzing

Challenges

EXERCISE 1

Connect the format to the disk. Write `pack_dir(path, compress=False)`, which archives the regular files of one folder (skipping sub-folders, in sorted order), and `unpack_to(buf, dest)`, which validates the whole archive first, refuses to overwrite any existing file, and only then writes the files. Use a temporary folder: pack a folder with two files and a sub-folder, unpack it elsewhere and compare, try to unpack twice, and show that a hand-built archive containing the name `../evil.txt` is rejected and writes nothing.

View solution

EXERCISE 2

Close the gap the fuzzer found. Define version 2 of TBX: the same layout with the version byte set to 2 and a 4-byte little-endian CRC-32 of everything before it added at the end. Write `write_v2` and a `read_any` that accepts both versions (checking the whole-file checksum first for version 2). Fuzz both versions with the same harness and show that version 1 accepts some damaged files while version 2 accepts none, and show what a one-bit change to a name does in each.

View solution

EXERCISE 3

Write a reusable `selftest(seed, rounds, mutations)` for the format: seeded random archives round-tripped both compressed and not, then thousands of randomly damaged copies (one to four byte changes, sometimes a truncation), counting round trips, accepted and rejected files and any crash by exception name. Show that the same seed gives the same report. Finish by reading a 50-million-byte bomb under `tracemalloc` and showing the peak memory stays small.

View solution

QUICK REFERENCE

- **Design first:** magic, version, flags with an unknown-bits rule, explicit sizes, a checksum, and written-down limits
- **Header:** `struct.Struct("<4sBBH")`; entry header: `struct.Struct("<BIII")`
- **Writer:** validate names, sizes and counts so it never produces what the reader refuses
- **Reader:** one gate (`take`), one error type (`FormatError`), check every claim before using it, reject trailing bytes
- **Compression:** `decompressobj().decompress(data, expected + 1)`, then compare length, `eof` and `unused_data`
- **Testing:** seeded round trips, hand-damaged files, a bomb, and a truncate-and-mutate fuzzer
- **A fuzzer's accepted cases:** read them; they point at fields the format forgot to protect
- **Evolving:** new flag bits and a new version byte, never a silent change of meaning

COURSE COMPLETE

You can now read a hex dump, decode integers of any width and byte order, describe a layout with `struct`, recognise files by their signatures, walk the chunks of real formats, pull apart flag bytes and nested records, defend a parser against hostile input, and design a format of your own. The next time you meet an unfamiliar binary file, the method is the same: find the signature, find the header, learn what each length and flag claims, and verify every claim before you trust it.